MASTER EN SCIENCES INFORMATIQUES

Conception d'un framework de mesures pour des prototypes d'applications réparties

Schmidt, Jean-Claude

Award date:
2009

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2008-2009

Conception d'un FrameWork de mesures pour des prototypes d'applications réparties

Jean-Claude Schmidt

Septembre 2009

Mémoire présenté en vue de l'obtention du grade
de Licencié en informatique.

Résumé

Les systèmes distribués font parties intégrantes de la programmation d'aujourd'hui, et la problématique des performances de ceux-ci existe pratiquement depuis leurs apparitions. La simulation de systèmes distribués est quand à elle souvent utilisée lorsque un architecte logiciel doit faire des choix technologiques et/ou de déploiement d'une application distribuées au moment ou l'on définit l'architecture logicielle. L'architecte logiciel doit souvent prendre de telles décisions sur base de sa connaissance du système et/ou sur sa propre expérience. La conjonction de ces deux points, la simulation et les performances de systèmes distribués, fait naître le besoin d'utiliser des outils d'aide à la décision spécifiques à ce domaine. Nous tentons de remédier à ce problème en proposant une boîte à outils offrant un certain nombre de fonctionnalités permettant de définir des prototypes de système distribué et de faire des mesures de performances, et ce dans un paradigme orienté objet.

Mots-clés : Programmation orientée objet, Java, RMI, Mesure, Système distribué, Simulation, Emulation, Performance, Composant, RMI.

Abstract

The distributed system are usefully all days and the problem of their performance exists since their appearition. The simulation of distributed system is often used when an software architect must make technologicals choices and/or distributed application deployement at the software design definition. The software architect must often to take such decisions on his system knowledge and/or on his own experience. The conjunction of both points, the simulation et the performance of the distributed systems, raises the need to used decision help tools for this domain. We attempt to propose a framework with functionality to define prototype of distributed system and to make performances measures, and this in Object-Oriented paradigm.

Keywords : Object Oriented Programming, Java, RMI, Measure, Distributed System, Simulation, Emulation, Performance, Component, RMI.

Avant-propos

Mes remerciements vont avant tout au professeur Englebert, mon promoteur, pour m'avoir aidé, soutenu et guidé dans mon travail ainsi que pour sa grande patience.
Merci à Mme Isabelle Linden pour ses encouragements lors de l'élaboration de ce mémoire et pour tout le travail qu'elle accompli au sein de la licence à horaire décallée.
Merci à Mme D'Udekem-Gevers pour la "somme relative aux mémoires", ce document est une aide précieuse pour la rédaction du mémoire.
Merci à Alain Biver, Philippe Deleuze et Pascal Gatellier, des collègues de travail, pour leurs encouragements et pour m'avoir allégé un peu mon temps de travail, ce qui m'a permis de rédiger ce mémoire.
Enfin, un merci tout particulier à mon épouse Nathalie pour m'avoir soutenu et encouragé à terminer mes études.

Accréditations techniques

Ce rapport de mémoire a été mis en page au moyen de LaTeX, un macro langage créé par Leslie Lamport pour le système de mise en page TeX créé par Donald E. Knuth. Pour toute information sur TeX et LaTeX , voir le site www.ctan.org.

Le texte a été rédigé avec *TeXnicCenter*, un environnement d'édition et de production de texte écrit en LaTeX, créé par Sven Wiegand en 1999 et maintenu par Tino Weinkauf, voir www.texniccenter.org pour plus d'informations.

Les figures ont été réalisées sur *Microsoft Office Visio 2003*, une suite applicative sous licence Microsoft.

Les captures d'écran ont été réalisées au moyen de *Snagit 3.2* de TechSmith corporation (www.techsmith.com), produit sous licence commerciale.

Le framework a été réalisé en *Java* java.sun.com avec un JDK 1.5 et développé sur la plate-forme *Eclipse 3.2*, un produit open source sous licence CPL (Eclipse Licence) créé par la société IBM et maintenu aujourd'hui par la fondation Eclipse www.eclipse.org.

Les diagrammes UML ont été créés avec EclipseUML Free Edition version 2.1.0 développé par la société Omondo (www.omondo.com).

Table des matières

Table des figures

CHAPITRE 1

Introduction

Les performances d'un système distribué (SD) peuvent-être fortement influencées par la manière dont la charge du système est répartie, cette répartition de charge (*Load Balancing*) permet d'améliorer les performances d'un système distribué en répartissant la charge parmi les différents noeuds [1] qui le compose.

Il existe différentes stratégies de load balancing :

1. Statique, c'est-à-dire que l'allocation de ressource se fait à la compilation et sur base d'une certaine connaissance du système mis en place ;
2. Dynamique (aléatoire, basé sur la sémantique, ...), c'est-à-dire que la répartition de charge se fait lors de l'exécution et en fonction d'un certain nombre de paramètres.

Ces différentes stratégies sont tantôt efficaces pour un modèle, par exemple avec de nombreuses petites requêtes ou avec moins de requêtes mais avec une taille de messages plus importante, mais il arrive parfois que celles-ci soient pénalisantes pour diverses raisons.

Tout développeur qui s'intéresse aux systèmes distribués se posera tôt ou tard ce genre de questions :

1. Est-ce que l'application est distribuée de manière optimale ?
2. Comment savoir si mon architecture distribuée est meilleure qu'une autre ?

L'objet du mémoire est donc de fournir des outils (*le Framework*) qui pourront apporter des éléments de réponses sur base d'intuitions dont on disposerait au moment de l'analyse du système distribué. L'idée est de pouvoir définir rapidement un

1. Un noeud est une machine physique dans un réseau

SD et de pouvoir faire des mesures de performances qui nous permettraient de faire des choix de design dans le SD.

Evidemment, nous ne voulons pas écrire une application distribuée complètement, c'est-à-dire que nous voulons décrire le fonctionnement ou plutôt simuler un SD qui serait une abstraction d'un système réel, c'est pour cette raison que nous parlerons de simulation de systèmes distribués.

La simulation des systèmes distribués est typiquement utilisée lorsque le système étudié n'est pas disponible pour la mesure ou lorsque le système ne possède pas de points de test pour mesurer chaque détail d'intérêt. La simulation est parfois le moyen le plus économique pour l'évaluation des systèmes distribués, elle sert à créer un prototype, qui reflète le fonctionnement du système, dont le temps développement est plus rapide qu'un système complet.

Même si le terme *simulation* est souvent employé, le terme le mieux approprié serait plutôt *émulation* :

- La simulation consiste à modéliser mathématiquement chacun des composants pour donner un résultat
- L'émulation quant a elle consiste simplement à reproduire les conséquences d'une architecture réseau : Les temps de réponse, la charge CPU ou la charge mémoire

Pour répondre aux questions posées ci-avant, il est donc nécessaire que nous puissions faire des mesures de performances et de pouvoir ensuite comparer celles-ci entre les différents architectures simulées.

Le framework proposé est, et reste, un outil pédagogique qui évoluera au fur et à mesure de nouveaux besoins.

Le présent document est structuré de la manière suivante :

- Le chapitre 2 pose les bases de ce travail, l'idée de départ
- Le chapitre 3 présente les hypothèses de travail ainsi que la démarche utilisée
- Le chapitre 4 présente un état des lieux au début de nos recherches
- Le chapitre 5 expose de manière générale les différents concepts mis en place dans le framework
- Le chapitre 6 détaille les composants du chapitre 5
- Le chapitre 7 expose les trois études de cas que nous avons étudiées et testées avec le framework
- Le chapitre 8 donne les conclusions et les pistes d'améliorations possibles

CHAPITRE 2

Idée de départ

L'idée de départ concernant ce mémoire est la suivante : *"J'aimerais simuler le comportement d'un SD et je voudrais faire des mesures de performances afin de pouvoir faire des choix d'architectures, comment faire ?"*.

Nous savons que pour faire fonctionner une application distribuée, nous avons besoin de plusieurs ordinateurs et nous devons pouvoir déployer les différents composants de celle-ci sur ces ordinateurs.
On va donc disposer d'un certain nombre d'ordinateurs (A , B, C), sur ces ordinateurs tourne une couche (un programme) que l'on va pouvoir configurer nous même. Ce programme tournera sur tous les ordinateurs sur lesquels l'application distribuée est censée pouvoir se déployer.

Imaginons le fonctionnement (simplifié) d'un système Peer-To-Peer (P2P) comme par exemple Gnutella. C'est un programme qui va se déployer sur les 3 ordinateurs et si on définit le comportement d'un programme Gnutella de manière très simple, ce programme va par exemple recevoir une requête et répondre (donc c'est synchrone).

1. A reçoit une requête,
2. il cherche localement dans sa BD s'il trouve quelque chose
3. et il demande aux autres de faire la même chose.

Sur la figure 2.1 la requête arrive sur A, elle se propage sur B puis sur C, et une fois que l'on détecte la boucle (retour sur A), le résultat retourne d'ou il vient.

Si on veut programmer cela un peu à la louche, on imagine sans problème qu'il nous faudrait sur A un service (par ex. "Request") qui reçoit en argument une requête

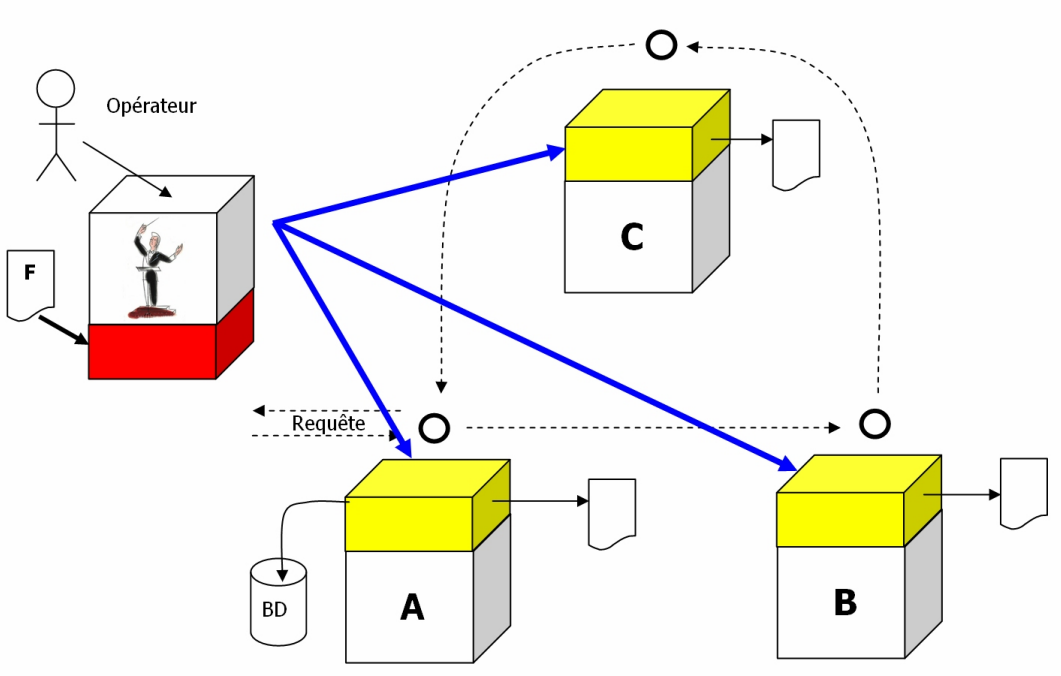

Figure 2.1: *Exemple d'un système Gnutella*

(une chaîne de caractères) et qui retourne une autre chaîne de caractères, le contenu de la chaîne de caractères n'a aucune importance ici.

```
//
service request (string a) {
        ...
        return string(2000);
}
//
```

Comme nous l'avons dit précédement, A doit faire une requête SQL pour aller vérifier que l'on a quelque chose dans la BD correspondant à la requête. On va regarder dans la BD si on trouve quelque chose et puis en même temps on va déléguer l'appel à un autre ordinateur (B).
On devrait donc faire une requête SQL mais pour rappel comme nous sommes dans de la simulation, ce qui nous intéresse ce n'est pas de faire fonctionner le système P2P mais bien d'en émuler le fonctionnement pour faire des mesures de performances.
Il nous faut donc simuler la requête et l'accès à la BD, pour ce faire on va remplacer la requête SQL par un appel disque [1] (disk(100,2)) avec 100 lectures et 2 écritures dans un simple fichier, ce qui permettra de simuler une charge (un accès) disque comme si une requête était réellement exécutée. Ensuite on va forwarder le message

1. il nous faudra donc des primitives pour émuler des lectures et des écritures sur disque

à un autre sauf si on est le dernier.

Nous pourrions dire aussi que dans 2/3 des cas on va forwarder et dans 1/3 des cas on va simuler quelque chose qui s'apparente à la réalité (c'est un peu comme jouer aux dés !). Ce ne sera pas vraiment la réalité, **c'est un abstraction de celle-ci**.

Donc nous aurions par exemple une instruction **IF** , mais comme on ne dispose pas des vrais arguments, des vraies conditions opérationnelles, on va remplacer l'expression booléenne par un tirage au sort qui va représenter la distribution de la variable booléenne.
On pourrait dire que dans le tirage au sort, dans 2/3 des cas c'est vrai et dans 1/3 des cas c'est faux, et c'est cela le jeu abstrait, c'est un test qui 2 fois sur 3 réussit.

```
        //
        service  request (string  a)  {
                if (66%)
                then {
                        X.request (a.size);
                }
                return string(2000);
        }
        //
```

Il faut retourner un résultat, et ce résultat sera par exemple un chaîne de caractère qui est de taille 2000. On construit ce string de 2000 caractères, peut importe son contenu, on va utiliser le fait d'envoyer cette chaîne pour simuler la charge sur le réseau.

Vu que nous retournerons toujours un résultat mais en plus dans 2/3 des cas on délèguera, il nous faudra donc connaître une référence d'un autre objet (par exemple X).

Nous aurons une procédure qui va faire des lectures et des écritures dans un fichier dont le contenu ne nous intéresse pas, on s'en servira juste comme fichier temporaire d'écritures et lectures.

Nous pourrions avoir également une instruction qui permettrait de simuler la charge du processeur.
Par exemple l'instruction ***cpu(100)*** prendrait en argument un indice de boucle, dans ce cas cela pourrait indiquer une boucle de 1 à 100 avec par exemple des fonctions mathématiques comme des sinus et/ou cosinus dans chaque itération, cela simulerait la charge du processeur.
On pourrait même faire plusieurs primitives comme par exemple ***cpuInt()*** avec des calculs sur des valeurs entières et ***cpuFloat()*** pour des réelles, ou même sur des chaînes de caractères comme ***cpuString()***.
Ce qui nous donnerait actuellement ceci :

```
        //
```

```
service  request (string  a)  {
        disk (100, 2);
        if (66%)
        then {
                X.request (a.size);
        }
        cpuFloat(100);
        return string(2000);
}
//
```

Revenons sur le problème du X !

Nous n'avons pas encore pris en compte le problème du chaînage des objets. Il va falloir donc avoir un autre service qui serait par exemple "Connect" qui manipulera des adresses ou des strings (IOR, adresse WEB).

```
//
service  connect (address x)  {
        X = x;
}
//
```

A ce stade, nous prenons comme hypothèse qu'avec notre langage, nous pourrons représenter un sous-ensemble de l'ensemble des programmes distribués, mais pas tout, il y a des cas qui ne pourront être représentés avec notre langage abstrait [2].

Les couches (en jaunes) sont des programmes que l'on démarrera manuellement. Ce programme sera à l'écoute, en attente, et il attendra qu'on le sollicite. Ces couches ne se connaissent pas, hors il est primordiale pour la délégation de la requête que ces couches se connaissent (dynamiquement ou statiquement), nous aurons donc besoin d'une autre couche pour orchestrer tout cela (couche rouge), un chef d'orchestre (CO).

Nous pourrions donc avoir besoin d'un autre langage, qui est un langage de programmation, ou plutôt un langage de déploiement.
On doit y définir les machines du système, les composants ainsi que la dynamique du système.

Nous pourrions définir tout cela dans le fichier de déploiement (F) dont le contenu serait semblable à ceci :

```
MACHINE
A : 1 ;
B : 2 ;
C : 3 ;

COMPONENT
C1, C2, C3 on A;
```

2. Ces autres cas pourraient faire l'objet d'autres mémoires

```
D1, D2 on B;
E1 on C;

DYNAMIC
C_Connect_B(C) (1)
B_Connect_A(B)
A_Connect_C(A)

A.Start() (2)
```

1. L'application C qui invoque "Connect" sur l'application B et on passe en argument l'adresse de C, de cette manière B apprend l'existence de C, et il connaît maintenant son adresse
2. Invocation du service "Start" sur l'objet A

Nous devons donc définir un nouveau service "Start", qui est une boucle qui ne s'arrête jamais, à 100% c'est toujours "True". La boucle invoque le service "request" sur l'adresse X avec comme argument une chaîne de caractères de longueur variable.

```
        //
        service  start () {
                while (100%) \{
                        X.requete (string(100)) ;
                }
        }
        //
```

Pour ce qui est de l'exécution du code, nous avons deux manières de procéder.

1) L'idée est de pouvoir spécifier une application de manière très rapide et de pouvoir faire des tests. Par exemple, si on a une dizaine de machines sur lesquelles tourne la couche en jaune, et que l'on a configuré ces machines pour qu'elles pointent toutes vers un fichier qui est sur un disque partagé sur un serveur, il suffira alors d'éditer ce fichier et de lancer une requête sur les différents serveurs pour les réinitialiser et de cette manière on a tout redéployé et on peux observer, mesurer, visualiser le résultat en temps réel, et donc on peut affiner l'architecture.

Ceci est le cadre idéal mais l'inconvénient est que cela nous oblige à définir un langage et de devoir écrire un parseur avec Lex et Yacc, etc.

2) Une autre façon de faire (plus rapide, on ira plus loin mais c'est aussi beaucoup plus lourd) c'est plutôt que de définir notre propre langage, c'est de créer une sorte de librairie dans laquelle on retrouverait des primitives. Et plutôt que de définir l'application abstraite dans ce langage, on utilisera un vrai langage comme Java, C++, et on simulerait l'application en invoquant des opérations de la librairie. Il suffirait d'inclure cette librairie dans un programme Java et faire des appels à cette librairie (cf. ci-après).

Exemple en CORBA (la syntaxe n'est pas respectée) :

On pourrait avoir par exemple une interface " Gnutella " dans laquelle j'aurais l'opération "Connect", "Start", et "Request" qui retourne une chaîne de caractères.

```
Interface Gnutella {
                Connect (in gnutella g)
                Start ()
                String Request (in string)
}
```

Maintenant implémentons par exemple en Java la classe Gnutella. Pour l'implémentation de la méthode ***request()***, au lieu d'implémenter pour de vrai, on va plutôt appelé une méthode ***"simul.disk()"*** , celle-ci va simuler la charge du disque et puis on peut aussi faire par exemple un IF ( ) et on appelle une autre méthode de la classe simul, par exemple ***proba(66)***, celle-ci retournera simplement True ou False avec 66 pourcents de chance et on appellera X.request, et ainsi de suite.

```
        //
        Connect ( in gnutella g ) {
                X=g
        }

        Request (string s) {
                simul.disk(1000,'w')
                if (proba(66)) {
                        X.request()
                }
                cpuFloat(100)

        }
        //
```

Donc ici on va aller beaucoup plus vite.

Ce sont 2 démarches intéressantes :

- La 1ère , avec un langage interprété, permettra de développer très rapidement. J'ai une idée, je prends mon fichier script et je l'écris très rapidement mais je n'ai pas toute la puissance d'un vrai langage
- La 2ème, c'est plus lourd, on doit l'interfacer avec des libraires Java, on a pas tout le confort mais on peut mélanger du concret et de l'abstrait, du vrai et du faux. Je peux écrire certaines partie en vraie et d'autres en abstrait dans des cas ou l'on aurait des inconnues dont on ne contrôlerait que quelques hypothèses, on peux mélanger la réalité avec le faux.

Ce sont 2 démarches orthogonales, c'est le dilemme efficacité contre confort et il est de fait difficile de concilier les deux.

Dans la première approche, il faudra probablement un langage de configuration et un langage pour le comportement. Il faut donc un langage du style XML (ou fichier INI).

Attention qu'il y aura toujours des choses que l'on ne pourra pas faire avec notre langage. Car par exemple on fait l'hypothèse que le nombre d'ordinateur sur lesquels

l'application serait déployée est connu. Il ne peut donc y avoir un nouvel ordinateur qui " apparaît " en cours d'exécution.

Autre exemple d'hypothèse à prendre, sur une machine on va prendre une optique qui est que lorsque l'on invoque une opération, il y a quelque chose qui s'exécute puis la machine est bloquée pendant l'exécution du service, ou on peut prendre plutôt l'optique qui serait du multithread.

On devra donc définir une politique, faire des choix simplificateurs qui seront quelques part des inepties par rapport à la réalité mais le but n'est pas de faire de la réalité mais bien de pouvoir valider rapidement un petit modèle, le plus important est que cela fonctionne avec un petit langage pas très riche.

Exemple :
Si on veut définir un client-serveur, et que les données connues sont du genre "on a 100 clients, 1 serveur, chaque client émet +/- 100 requêtes par jour, etc.", et bien on va pourvoir simuler ce cas et voir un peu ce qui se passe grâce à notre framework.

Pour cet exemple, on imagine sans problème que l'on pourra faire des mesures de performances, mais que veut-on mesurer ?

- Le temps d'exécution de chaque requête ?
- La somme de tous les accès au service ?

Devra-t-on déclencher un chronomètre (chrono) et puis faire des mesures à des instants précis ? Ou est-ce le système qui prend des mesures indépendamment ou sur base de choix pré-établis, on mesure le nombre d'invocation ou des mesures globales ?

On peut aussi laisser le choix à l'utilisateur de mettre des choses dans son code comme par exemple chrono.start(toto), chrono.end(toto), chrono.start(titi), chrono.end(titi), et cela mesurerait les temps cumulés des chronomètres dans un fichier et une fois la simulation terminée, on regarde ce fichier et on pourrait voir par exemple :

```
toto : 100 sec
titi : 40 sec
```

Ce qui est demandé c'est de faire des choix, par exemple compter le nombre de caractère dans une page html n'est pas très intéressant, par contre le nombre d'invocation par requête à plus de sens.

Si on mesure le temps que prend une requête entre le moment ou on l'invoque et le moment ou on reçoit le résultat, ce temps risque d'être différent si on refait la même mesure avec la même requête juste après. Cette constatation nous informera que le système n'est pas prévisible et cette information sera déjà intéressante.

Si on observe le temps sur des grandes périodes de simulation, pas sur 1 minute mais plutôt sur 1 heure, et sur cette heure on peut avoir une moyenne de 9 à 10 secondes pour une requête. On aurait donc une impression qui est plus fiable que si on ne savait rien.

Revenons sur le fichier de configuration (F).

Nous avons la liste des machines touchées par le système :

```
MACHINE
A : 1 ;
B : 2 ;
C : 3 ;
```

Nous avons définit les composants :

```
COMPONENT
C1, C2, C3 on A;
D1, D2 on B;
E1 on C;
```

Pour que les composants puissent communiquer ensemble, il faut leur indiquer la dynamique du système :

```
DYNAMIC
C1_Connect_C2(C1)

C1.Start()
```

Ce qui veut dire dans ce cas que le composant C1 appel la méthode ***Connect()*** sur le composant C2 et lui passe comme argument son adresse, l'idée étant que C2 découvre l'existence de C1 et puisse communiquer avec C1.

Ces informations se trouvent dans un fichier (ou autre comme par exemple une classe java) qui posera l'environnement du simulateur.

Une fois que ce fichier est écrit, on va appeler une méthode qui va mettre tout le système en marche. Cela veut dire que si l'on fait cela, du côté du chef d'orchestre on va envoyer un message à la couche ou se trouve C1, c'est-à-dire la machine A, "Attention C1, tu dois faire ceci !".

Le chef d'orchestre jouera donc le rôle de l'opérateur derrière chaque machine. Le chef d'orchestre est donc un programme qui est capable de lire (ou d'exécuter) ce fichier et de réagir en fonction de ce qui se trouve dans celui-ci.

Dans le figure 2.2, nous pouvons voir une ébauche de ce que serait l'arbre abstrait de notre langage et l'appel aux primitives (par exemple écrites en Java) qui permettrait de simuler notre système dans la solution de l'interpréteur Lex et Yacc.

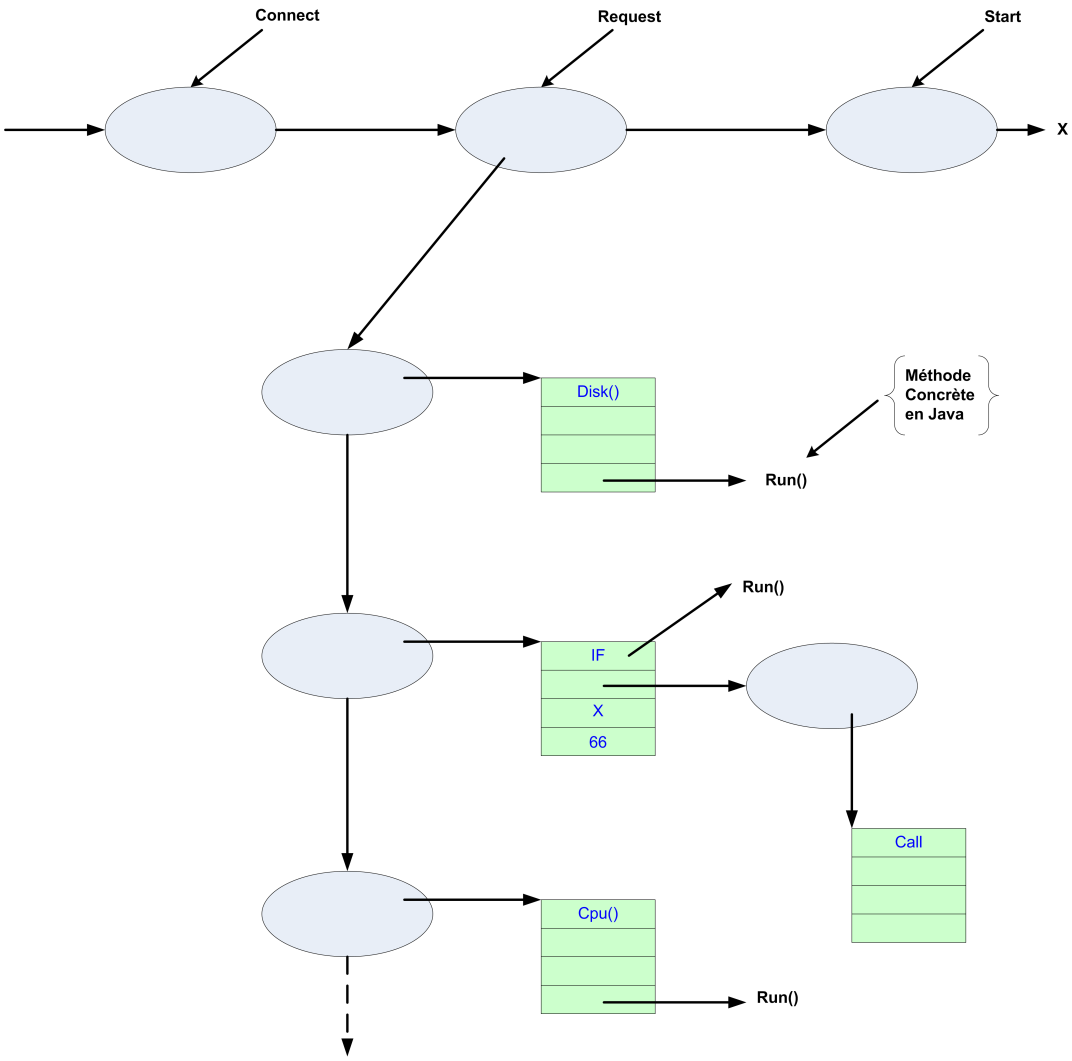

Figure 2.2: *Représentation de l'arbre abstrait*

CHAPITRE 3

Hypothèse de travail

Pour élaborer ce travail, nous avons étudier des cas de systèmes distribués allant du simple serveur de page web jusqu'au système plus complexe comme le TP MDL de rendu d'images 3D, en passant par un système P2P (simple) comme Gnutella.

Comme dans tout travail de recherche, nous devons faire certaines hypothèses de travail.

Notre framework permettant de simuler des prototypes de SD et celui-ci utilisant principalement la notion de "Machine", nous posons notre première mais important hypothèse de travail à savoir que le système proposé est un système clos, c'est-à-dire que le nombre de machines est connu et une fois le système démaré aucune nouvelle machine ne peut apparaître et/ou disparaître.

Bien évidemment le nombre de machines peut varier mais pas de manière dynamique et pour cela il faudra modifier les différents scénarios que l'on peut exécuter et donc cette prise en compte d'une nouvelle machine ne sera effective qu'a l'exécution du nouveau scénario.

Cette hypothèse est également valable pour le nombre de composants par machine, étant donné que la composition du système est définie dans un fichier dit de "scénario", les composants par machine font également parties de ce système considéré comme clos.

Une notion importante est qu'avec notre système,nous ne pourrons pas tout simuler, nous ne pourrons représenter qu'un sous-ensemble de systèmes distribués.

Qu'entend-t-on par "simuler" des choses ?, il est également primordial d'expliquer ce que l'on entend par "simuler" des "prototypes" de SD.

La notion de simulateur est ici employée pour exprimer le fait que l'on ne pro-

gramme par réellement ce que l'on veut représenter, par exemple pour décrire un système P2P il faut gérer les accès à une base de données, mais tout cela est bien trop lourd et nous sommes dans un contexte de simulation, donc plutôt que de faire des accès à une vraie base de données on va utiliser des méthodes qui lisent et qui écrivent dans des simples fichiers textes, c'est beaucoup plus simple à gérer.
L'idée de base est que l'on veut écrire un petit système de manière simple et rapide, donc on ne veut pas gérer une base de données alors que l'on voudrait simuler un système P2P qui accède réellement à une BD. Etant donné que l'on veut surtout simuler des charges réseaux et/ou processeurs, nous utiliserons aussi d'autres mécanismes comme par exemple des primitives qui feront un réel travail au niveau CPU et cela permettra simplement de simuler un charge.

La notion de "prototype" est exprimée par le fait que nous définisons un SD ou un sous-ensemble de manière rapide et un peu à la louche, nous ne définissons donc par réellement un SD de manière stricte et formel, il s'agit donc plus d'un prototype de SD.

Bien que normalement on ne se préoccupe pas du langage de programmation lors de l'analyse conceptuelle, la question de l'implémentation s'est cependant vite posée, en effet il est important de rappeler que nous proposons un outil pédagogique et que nous avons fait comme hypothèse que nous partons du concret vers l'abstrait. Dès lors nous avons très tôt fait de petits tests pour valider certains concepts et le choix du middleware s'est porté sur RMI pour sa simplicité et pour sa facilité de mise en oeuvre. Quand au langage de programmation, qui rappelons ne dépend par forcément du middleware, le choix s'est porté sur Java tout simplement parce que RMI fait partie intégrante de Java et n'existe que pour Java au contraire de CORBA qui lui peut utiliser du Java ou un autre langage comme le C++.

Le système proposé doit rester simple tant sur les concepts que sur l'utilisation car il doit permettre de valider rapidement un petit modèle de système distribué.

Quant à la gestion des appels concurrents de méthodes distantes, il faut savoir que Java/RMI crée automatiquement un thread lorsque deux clients accèdent en méme temps à une méthode distante, et donc de ce fait l'utilisateur ne doit pas gérer la concurrence lui-même car Java/RMI le fait de manière implicite.

Par contre, et cela est très important de l'avoir à l'esprit lorsque l'on écrit des composants, c'est que Java/RMI ne gère pas la synchronisation des accès en lecture/écriture aux attributs d'objets. Si un composant possède des attributs propres et que ceux-ci sont modifiés par une des ses méthodes déclarée comme distante, alors il devra lui-même ajouter le mot ***Synchronized*** dans la déclaration de la méthode, en gardant bien à l'esprit que les performances s'en ressentiront, mais cela peut

aussi faire l'objet de tests pour des mesures de performances, donc le choix de la synchronisation incombe au développeur du système distribué.

CHAPITRE 4

Etat de l'Art

Tout travail de recherche commence par faire l'état de l'art, c'est-à-dire faire un état des lieux de l'existant par rapport au domaine étudié.

Au début de notre travail de recherche, il n'existait rien de semblable et dès lors nous sommes partis d'un feuille blanche avec comme seul point de départ l'idée de pouvoir écrire un petit système distribué de manière rapide et simple afin de pouvoir faire des mesures de performances, mesures qui comme nous le verrons plus loin pourront-être de plusieurs types.

En 2005, soit un an après le début de notre travail, un article a été publié par Grundy Cai et Liu *[GCL05] "'SoftArch/MTE : Generating Distributed System Test-beds from High-level Software Architecture Descriptions"'*. Nous en avons fait une analyse à posteriori et il en résulte que certains des concepts mis en place dans notre framework sont aussi présent dans SoftArch/MTE, ce qui nous conforte dans notre solution.

Dans *[GCL05]*, il est dit que "'La plupart des spécifications de système distribués ont des exigences de performance, par exemple le nombre de transactions requis par seconde doit être supporté par le système distribué. Cependant, la détermination de performance dans des systèmes distribués complexes durant leur développement est un vrai défi."'
SoftArch/MTE est donc un outil de description et de génération de code permettant de faire fonctionner un système distribué et de faire de mesures de performances.

Grâce à SoftArch/MTE, un architecte logiciel pourra définir, avec un haut niveau d'abstraction, un système distribué de manière assez complète grâce à un outil

graphique. Il pourra notamment définir les clients, les serveurs, les machines faisant parties de son système, il pourra aussi choisir le middleware pour la partie communication (RMI, CORBA, DCOM, etc.).

A la différence de notre framework, les bases de données utilisées dans SoftArch/MTE ne sont pas remplacées par autre chose, ce sont de vraies BD avec de vraies requêtes SQL.
En ce qui concerne les résultats de tests, ils sont retournés à l'application principale de manière automatique pour être sauvegardés et éventuellement affichés dans des graphiques, les courbes de résultats sont générées par Microsoft Excel tout comme dans notre framework mais les fichiers ne demandent pas d'interventions manuelles comme dans notre solution.

Tout comme dans notre framework, l'architecte logiciel doit donner certaines propriétés du système : Les clients, le serveur, la base de données (pas dans notre framework), les machines avec leur adresse, le nombre de requête qu'un client fait ainsi que leurs fréquences [1] (par exemple 1000 fois en continu, toutes les 25 ms, etc.). Les requêtes BD peuvent aussi être précisées, leurs nombres de lignes retournées ou par exemple le type de requête (select, insert, etc.) peut-être choisi par l'architecte logiciel.

Toutes ces informations sont utilisées pour générer des fichiers XML qui représentent le système que l'on veut simuler. Ces fichiers sont traités par des feuilles de styles (XSLT) et ce afin de générer les vrais fichiers qui seront exécutés. Il exite plusieurs feuilles de styles en fonction du langage choisi par l'architecte (Java, C++, Delphi), ceux-ci génèrent aussi les interfaces utiles pour COM et CORBA.
Grâce à un agent de déploiement (écrit en Java/RMI), tous les fichiers sont déployés automatiquement sur toutes les machines du système, ce qui n'est pas le cas de notre framework ou le déploiement est manuel.
En ce qui concerne les types de mesures, SoftArch/MTE propose les suivantes :

- Le nombre d'invocation d'une requête distante
- Le temps d'appel d'une requête (total et moyenne)
- Le nombre d'accès à la base de données
- Le nombre de mises à jour effectuées en BD

Les deux premières sont disponibles dans notre framework.

Enfin les résultats peuvent-être visualisés dans des fichiers Excel et il est possible aussi de donner une version à chaque fichier de résultats afin de pouvoir faire des comparaisons entre versions.

1. Dans notre framework la fréquence des requêtes est précisée selon l'implémentation des composants

SoftArch/MTE est donc un outil qui peut-être utile pour un architecte logiciel lorsque l'on veut faire des mesures de performances, mais il ne simule pas un système distribué, il génère des fichiers de tests dans un vrai langage concret choisi par l'architecte logiciel et ceux-ci peuvent, à la demande, effectuer des mesures de performances.

CHAPITRE 5

Présentation conceptuelle

Ce chapitre présente les concepts mis en place pour implémenter le framework, de manière générale nous présentons les grands composants du système à savoir :

1. La fabrique de composant
2. La dynamique du système
   (a) Le Chef d'Orchestre
   (b) Les scénarios
3. La libraire de simulation
4. Le système de mesures
5. Les fichiers résultats

Nous commençons par faire un petit rappel de notions de bases notamment sur le fonctionnement de RMI.

5.1 Définitions

5.1.1 Système distribué

Un système distribué est un ensemble d'ordinateurs indépendants qui apparaît pour ses utilisateurs comme un seul système cohérent, chaque élément du système distribué est appelé aussi un noeud ou dans notre cas une machine.

5.1.2 Applications distribuées

Une application distribuée est une application dont les classes sont réparties sur plusieurs machines différents. Dans de telles applications, on peut invoquer des méthodes à distance. Il est alors possible d'utiliser les méthodes d'un objet qui est situé sur autre machine que la machine locale.

Depuis longtemps déjà on peut faire de l'invocation à distance avec le langage C en utilisant RPC (*Remote Procedure Calls*), mais RPC n'est pas orienté objet et le paradigme utilisé dans notre framework est un paradigme orienté objet puisqu'il s'agit de RMI (*Remote Method Invocation*). Ce que RMI apporte de plus par rapport à RPC est le fait que non seulement il est possible d'envoyer les données d'un objet, mais il est possible aussi d'envoyer ses méthodes et donc un objet complet. Cela se fait grâce à la sérialisation des objets (voir *Erreurs courantes avec RMI*)

Une autre technologie utilisée pour les applications distribuées est CORBA (*Common Object Request Broker Architecture*) qui est un standard d'objet réparti. CORBA a été conçu par l'OMG (*Object Management Group*), un consortium regroupant plusieurs entreprises. Le grand intérêt de CORBA est le fait qu'il fonctionne sous plusieurs langages et donc on peut l'utiliser avec autre chose que java, mais à l'inverse de RMI, il est plus lourd à mettre en place.

5.2 Fonctionnement de RMI

Etant donné que nous utilisons RMI, il nous semble intéressant de rappeler son principe de fonctionnement.

Pour rappel, RMI est un framework qui n'existe que pour le langage Java et donc si nous voulons utiliser RMI pour sa simplicité de mise en oeuvre, nous devons implémenter notre code en Java.

Le framework RMI nous fournit un moyen d'accéder aux objets appartenant à une JVM (*Machine Virtuelle Java*) distante et nous permet d'appeler leurs méthodes.
L'architecture de RMI est basée sur un concept fort : la définition du comportement et l'exécution de ce comportement sont des concepts séparés. La définition d'un service distant est codée en utilisant une interface Java et l'implémentation de celui-ci est codée dans une classe. Il est donc important de savoir qu'en RMI les interfaces définissent le comportement et les classes Java définissent l'implémentation.

RMI supporte deux types de classe qui implémentent la même interface, la première est l'implémentation du comportement et se trouve côté serveur et la seconde

agit comme un proxy pour le service distant et se trouve côté client.

Un programme client crée des appels de méthodes sur le proxy, RMI envoie la requête à la JVM distante et la transfère à l'implémentation.

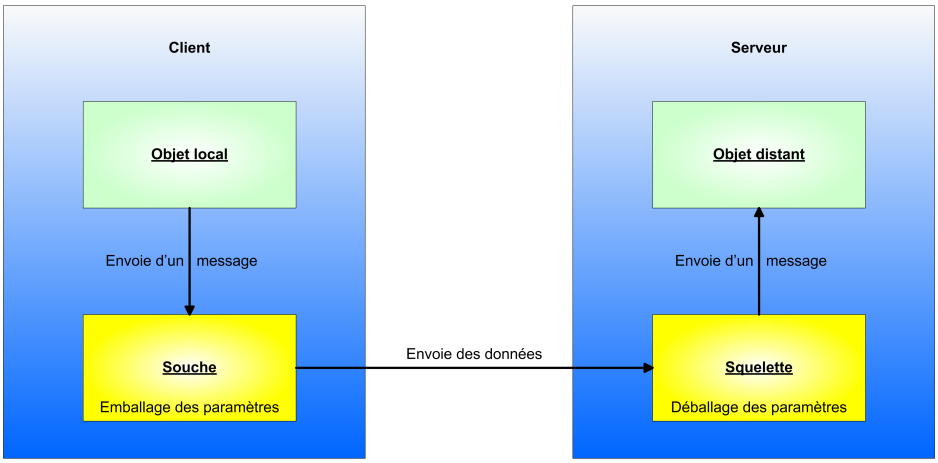

Figure 5.1: *Invocation d'une méthode distante*

Les valeurs de retours fournies par l'implémentation sont retournées au proxy puis au programme client.

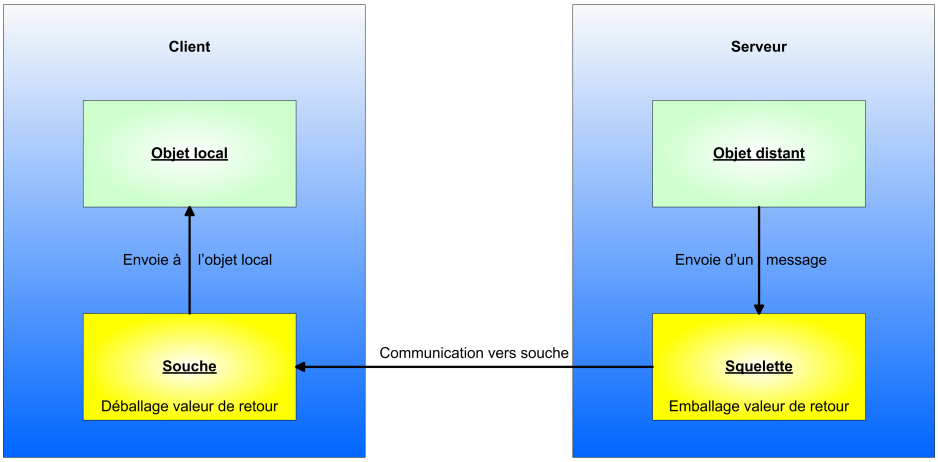

Figure 5.2: *Renvoie de la valeur de retour*

L'objet client se comporte comme si c'était lui qui exécutait les appels de méthodes à distance. En fait, il appelle les méthodes sur un objet "'proxy"' local qui gère tous les détails de bas niveau de la communication sur le réseau et donc l'appel

est complètement transparant pour le programmeur.

Donc le rôle du proxy distant est de jouer le rôle de représentant local d'un objet distant, le représentant local n'est autre qu'un objet sur lequel vous pouvez appeler des méthodes locales qui seront relayées à l'objet distant.

RMI est basé sur le pattern "'Proxy Distant"', celui-ci fournit un remplaçant à un autre objet, pour en contrôler l'accès.

Une autre information qui peut, comme nous le verrons plus loin, avoir son importance, est que RMI fournit un ramasse miette distribué (*Distributed Garbage Collector*), son intérêt est le même que le ramasse miette local. Il sert à supprimer les objets distants qui ne sont plus référencés par des clients. Il est activé lorsque plus aucun client ne possède de souche de l'objet distant, et c'est en cela que réside comme nous le verrons plus loin une faille dans le système.

5.2.1 Procédure de déploiement

Voici les différentes étapes à mettre en oeuvre pour la création de service distant. Autrement dit les étapes nécessaires pour prendre un objet et en faire un objet dont les méthodes peuvent-être appelées à distance par un client.

1. Créer une **Interface distante** : l'interface distante définit les méthodes qu'un client peut invoquer et doit hériter de l'interface *Remote* et toutes les méthodes utilisables à distance doivent pouvoir lever des exceptions de type *RemoteException*
2. Créer une **Implémentation distante** : c'est la vrai classe, celle qui définit l'implémentation réelle des méthodes définies par les interfaces distantes
3. Générer les **Souches/Stub** et les **Squelettes/Skeleton** : ce sont des classes utilisées par le client (souche) et par le serveur (squelette) [1], ces classes sont générés automatiquement par l'outil rmic de Java [2]
4. Exécuter le **Registre RMI** (rmiregistry) : il s'agit d'un annuaire, semblable à un annuaire téléphonique, c'est dans celui-ci que le client va chercher l'objet dont il veut invoquer les méthodes
5. Lancer le **Service Distant** : la classe d'implémentation crée une instance du service et l'enregistre dans le registre RMI et donc elle met ce service à disposition des clients

1. Depuis la version 1.2 de Java, le *skeleton* n'existe plus, seul le *stub* est nécessaire du côté client et du côté serveur
2. Il existe, et nous l'avons utilisé dans Eclipse, un plugin qui gère le tout de manière automatique *Genady's RMI plugin for Eclipse*

5.2.2 Erreurs courantes avec RMI

Les trois erreurs principales que l'on commet avec RMI sont :

1. Oublier de lancer rmiregistry avant de démarrer le service distant
2. Oublier de rendre les arguments et les types de retour sérialisables
3. Oublier de copier la classe souche côté client

5.3 La fabrique de composants

Nous devons permettre la création d'instances multiples d'une classe X sur les machines faisant parties du système. Comme nous avons bien des machines physiques différentes il faut savoir que nous ne pouvons pas utiliser le *new* de Java tel quel, car celui-ci ne gère que la mémoire locale, dans notre cas celle du CO. La solution à ce problème est l'utilisation d'une fabrique d'objets distants, celle-ci va créer localement (sur les machines distantes) les instances de X (en utilisant par exemple ***new X()*** côté serveur).

Un des grands composant du système est donc la fabrique de composants. La base même pour que le système fonctionne est que nous devons pouvoir créer des composants (*Component*) sur différentes machines (*Machine*) et que ceux-ci puissent communiquer entre-eux. Il est clair également que nous ne voulons pas, puisque nous sommes dans un contexte de simulation, écrire tout le code pour par exemple créer les composants.
Nous voulons à tout prix éviter que la personne qui veut décrire son SD soit obliger de faire un peu comme ceci :

```
//
Client c1 = new ClientImpl();
Client c2 = new ClientImpl();
//
```

De ce fait nous devons utiliser une autre méthode qui permettra de demander aux machines distantes de créer des composants, l'idée est que nous voulons juste donner un nom de composant à la fabrique et celle-ci fera tout le reste. Le CO va créer des machines (*Machine*) et ensuite va demander à ces machines de créer des composants (*Component*) en leurs spécifiants le nom de ceux-ci. Dans *[FF05]*, il est un pattern qui convient parfaitement à ce genre de concept, le pattern *Factory* (Fabrique), celui-ci est utilisé pour la création d'objet.
La fabrique de composant sera distante par rapport au CO et de plus, comme la fabrique doit pouvoir créer plusieurs types d'objets (Client, Server, ...), nous utilisons l'API *Reflection* (*java.lang.reflect*) de Java, cela permet d'instancier une classe sur base de son nom et non plus en faisant un simple appel à par exemple ***new ClientImpl()***.
De manière simple, le paquetage *java.lang.reflect* permet l'introspection en rendant possible l'accès aux classes, à leurs champs, méthodes ou encore constructeurs, et à toutes les informations les caractérisant, même celles qu'on pensaient inaccessibles. L'introspection nous permet de découvrir dynamiquement les informations d'une classe sans connaissance préalable du code source de celle-ci. Ce mécanisme introspectif est généralement utilisé par des outils de développement IDE (*Integrated Development Environnement*).
Le pattern *Factory* combiné à l'introspection Java va donc nous permettre de créer une fabrique de composants assez générique pour permettre de créer des composants

(objets) distants de manière simple juste en se basant sur le nom du composant. Une règle est cependant mise en place[3], nous passons le nom du composant à créer et la fabrique ajoutera systématiquement la chaîne "'**Impl**"' au nom de composant. Par exemple pour demander à la fabrique de créer une instance du composant "'ClientImpl.java"', nous faisons juste un appel comme par exemple ***Machine.createComponent("'Client"')***.

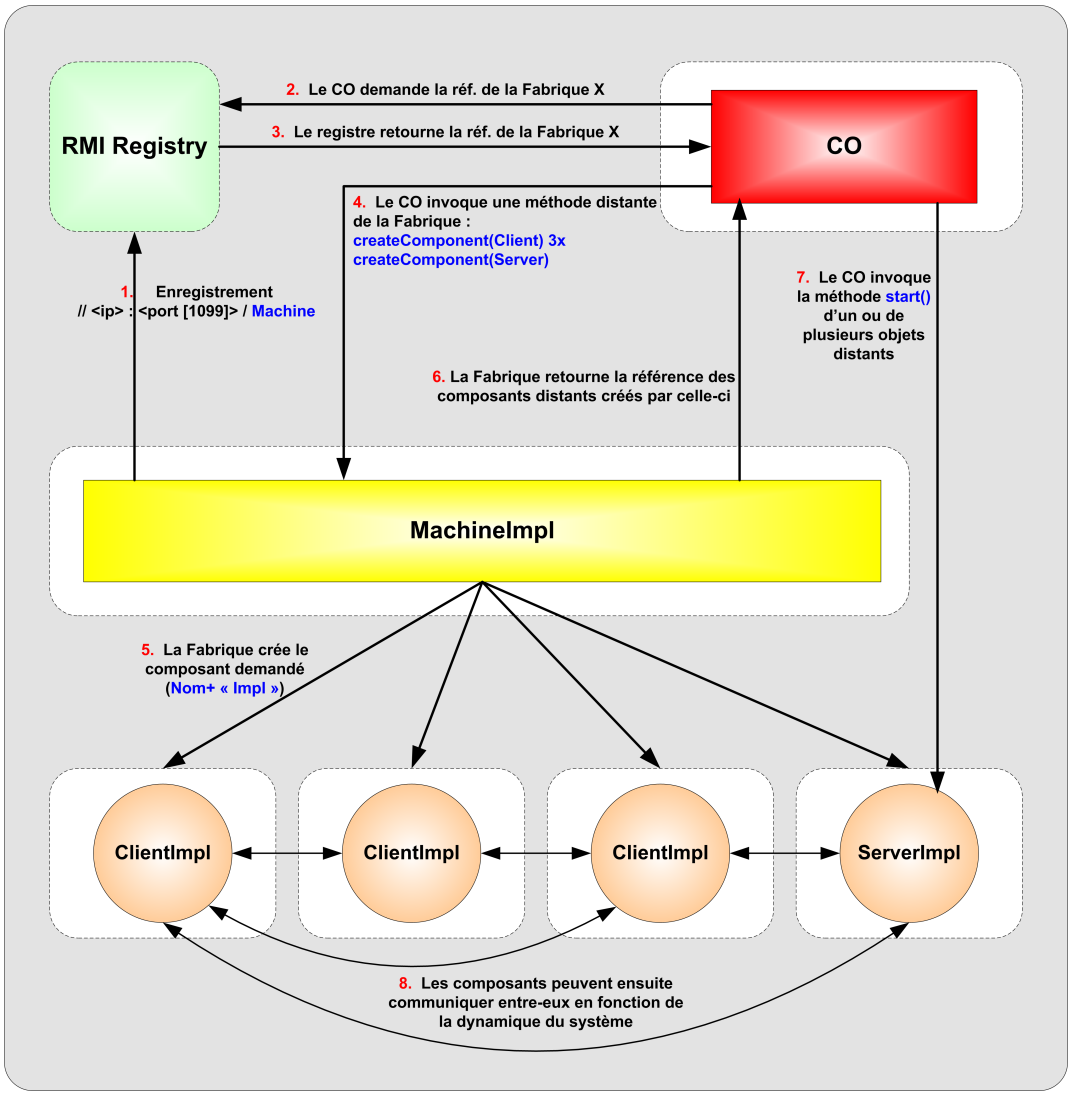

Figure 5.3: *Fonctionnement de la Fabrique de composants*

3. voir la partie ***Implémentation d'un composant***

Sans rentrer dans le détail de l'implémentation, voici le principe de fonctionnement de la fabrique :

1. La fabrique d'objet s'enregistre au niveau de la Registry et donne ainsi accès à ses méthodes distantes
2. Le CO se connecte au registre et demande la référence de la fabrique
3. La référence de la fabrique est retourné au CO, il pourra dès lors appelé une méthode distante de la fabrique
4. Le CO demande à la fabrique de créer des instances de classes qui sont passées en paramètre
5. La fabrique instancie les classes via l'API reflection
6. La fabrique retourne la référence des composants distants créés par celle-ci
7. Le CO peut invoquer les méthodes déclarées comme distantes (par ex. start()) sur les références retournées par la fabrique
8. Le code des composants peut s'exécuter et les composants peuvent dès lors communiquer entre-eux

5.4 La dynamique du système

La dynamique du système, ou chaînage d'objet, permet de préciser comment les objets peuvent communiquer entre-eux (le lien entre les composants), c'est-à-dire que l'on spécifie qu'un composant A doit faire appel à une méthode d'un autre composant B et non pas C. L'idéal est que se ne soit pas le composant qui recherche lui-même le composant suivant (celui dont il doit faire appel) parce que cela augmente la complexité du composant que l'on implémente, mais cela reste possible. Le CO doit pouvoir aussi pouvoir préciser cela au démarrage de l'application.

Pour ce faire, deux concepts sont mis en place pour réaliser cette dynamique :

1. Un objet ***Parameters()*** qui est passé en paramètre aux composants du système, cet objet (*sérialisé*) est en fait une table de *Hachage* (HashTable) qui contient une liste d'objets avec pour chacun d'eux une clé qui permet de les identifier et/ou d'obtenir leur référence
2. Un service offert par chaque composant, à savoir une méthode ***connect()*** qui permet de préciser à un composant quel est le suivant, autrement-dit quelle est la référence de l'objet distant dont le composant doit appeler une méthode

Passer une table de hachage comme paramètre est très pratique car nous pouvons y mettre une liste d'objet très facilement, à condition de bien identifier chaque élément de la table par une clé unique. Une table de hachage est une structure de données qui permet une association clé-élément, on accède à chaque élément de la

table via sa clé. La table de hachage est en fait un tableau ne comportant pas d'ordre.

Nous pouvons par exemple mettre la référence d'un objet distant en lui associant une clé (par exemple son nom) et il suffirait dans l'implémentation du composant de rechercher la référence de l'objet sur base de la clé (grâce à la méthode ***resolve(key)***, ce qui permettra de chaîner les objets entre-eux.
Voici un exemple de la manière d'utiliser la table de hachage :

```
//
...

public void start(Params p) throws RemoteException {
        super.start(p);

        w1 = (Component) parameters.resolve("w1");
        w2 = (Component) parameters.resolve("w2");
        w3 = (Component) parameters.resolve("w3");

}

public void request(String s) throws RemoteException {

        // Création de s1, s2 et s3
        ...
        // Appel distant de la méthode request()
        w1.request(s1);
        w2.request(s2);
        w3.request(s3);

}
...
//
```

Quant à la méthode ***connect()***, elle permet au CO de préciser le chaînage d'objet et donc ce n'est pas dans le code des composants qu'il faut rechercher le prochain. Ceci est surtout utile pour des liens de type client-serveur entre objets, de plus l'implémentation du code est un peu plus facile qu'avec la table de hachage. Il est en effet assez facile pour le CO de faire par exemple ***connect(C1,C2)*** ou ***connect(S)*** afin de préciser à l'objet C1 qu'il doit faire appel (le cas échéant) à l'objet C2, quant au deuxième exemple il précise que tous les objets doivent se connecter à l'objet S, ceci est très utile dans le cas de n clients se connectant tous au même objet de type *Server()*

Exemple avec table de hachage :
Côté CO :

```
Machine M1 = createMachine("M1", "127.0.0.1");
Machine M2 = createMachine("M2", "127.0.0.1");

Parameters paramS = new Parameters();
// Creation de 1 composant de type "Serveur" sur la machine M1
Components S = Components.createComponents(M1, 1, "component.
    Serveur", paramS);
```

```
        Parameters paramC = new Parameters();
        // Ajouter la référence du 1er composant de S avec "nextC"
            comme identifiant
        // C'est-à-dire que nous passons à tous les objets de C la
            référence du "serveur"
        paramC.add("nextC", S.get(1));
        // Creation de 1 composant de type "Client" sur la machine M2
        Components C = Components.createComponents(M2, 1, "component.
            Client", paramC);

        // Appel à la méthode startOneway du CO qui délèguera aux
            composants de la collection S
        S.startOneway();
        // Appel à la méthode startOneway du CO qui délèguera aux
            composants de la collection C
        C.startOneway();
```

Côté composant :

```
public synchronized void start(Params p) throws RemoteException {
...
        // Get the remote object identified by the key
        Serveur server = (Serveur) p.resolve("nextC");
        // Call the remote object
        server.request(...)
...
```

Exemple avec méthode connect() :
Côté CO :

```
...
C.connect(S.get(1)); // Tous les objets de C se connectent à S.get(1)
    autrement-dit au Serveur
C.connect(1,2);          // Autre exemple, ici le 1er composant de C se
     connecte au 2ème de C
...
connect(C.get(1), S.get(1));  // Autre exemple pour préciser que C(1)
    doit se connecter à S(1) via connect() de la chorégraphie
...
```

Côté composant :

```
...
if (proba(33)) {
        // appel au composant suivant en lui passant une chaîne de 100
            car.
        nextComponent.request(createString(100));
} else {
        // appel au composant suivant en lui passant une chaîne de
            20000 car.
        nextComponent.request(createString(20000));
}
...
```

5.4.1 Le chef d'orchestre

Comme nous l'avons précisé au début de ce document, un élément important dans notre système est ce que l'on appelle *le chef d'orchestre*, c'est lui qui initie le système.

Si l'on fait une analogie entre le monde de la musique et notre framework, un chef d'orchestre fait jouer une partition de musique par des musiciens, dans notre cas le chef d'orchestre va exécuter un scenario (la partition de musique), celui-ci sera joué par un joueur de scénario (par la classe *ScenarioPlayer*) via une chorégraphie (par la classe *Choreography*) .

C'est également le chef d'orchestre qui peut interrompre le fonctionnement du système, pour cela il dispose de deux mécanismes :

1. Il appelle la méthode stop() sur chaque Machine, reste aux machines à appeler la méthode stop() de chaque composant définit sur celle-ci via leur collection de composants
2. Il appelle lui-même la méthode stop() des composants car il possède lui aussi une ou plusieurs collections représentant tous les composants du système distribué, et ce pour toutes les machines.

La question que l'on peut aussi se poser est la suivante : *"'Comment faire en sorte que le CO arrête le système après disons 1 minute de fonctionnement ?"'*.
Nous avons opté pour la solution la plus simple à savoir une méthode pause() qu'il suffit d'appeler en précisant la durée de la pause (en msec).

Exemple de scénario qui laisse fonctionner le système pendant 1 minute :

```
...
S.start();
...
C.startOneway(); // Important pour rendre la main au scénario
...
pause(60000); // Wait 60 sec. the unit is the millisecond
// Stop the system
try {
        M2.stop(); // Firstly stop the Clients
} catch (RemoteException e) {
        e.printStackTrace();
}
try {
        M1.stop(); // Secondly stop the server
} catch (RemoteException e) {
        e.printStackTrace();
}
```

5.4.2 Les scénarios

Les scénarios sont des fichiers écrits en Java qui définissent le nombre de machine à créer, les composants par machine, la dynamique du système, c'est également dans

ce fichier que les méthodes ***start()*** des composants sont appelées.
Les scénarios sont joués par la classe *ScenarioPlayer*.
Les scénarios sont des classes Java qui doivent étendre la classe *Choreography()* et qui sont exécutées par l'API reflection, c'est-à-dire que le joueur de scénario va instancier la classe sur base de son nom, comme pour la fabrique de composants.
Voici un exemple de scénario :

```
public class scenario_1 extends Choreography {

        public scenario_1() {

                // Création d'une machine
                Machine M1 = createMachine("M1", "127.0.0.1");

                // Creation de 1 composant de type "Serveur" sur la machine M1
                Components S = createComponents(M1, 1, "component.Serveur",
                    null);

                // Creation de 100 composants de type "Client" sur la machine
                    M1
                Components C = createComponents(M1, 100, "component.Client",
                    null);


                // SYSTEM DYNAMICS
                // All the C component must be connected to component "S.get
                    (1)"
                C.connect(S.get(1));


                // Start the components
                S.start();
                C.startOneway();


                pause(30000); // Wait 30 sec.
                try {
                        M1.stop();
                } catch (RemoteException e) {
                        e.printStackTrace();
                }
        }
}
```

5.5 La librairie de simulation

La librairie de simulation est au départ une classe Java statique qui contient des méthodes qui permettent de simuler des choses (BD, charge CPU, etc.). La librairie a cependant été scindée en deux, à savoir :

1. Une librairie statique (non instanciable) qui reprend des méthodes comme *proba()*, *cpuFloat()*, etc., cette classe est nommée *Tools.java*

2. Une classe instanciable pour la partie accès BD (lecture/écriture dans des fichiers), cette classe est nommée *Simul.java*

La raison à cela est qu'au départ la création du fichier (son nom) se faisait dans la méthode *statique* disk(), et celle-ci devait-être déclarée ***Synchronized*** pour éviter les conflits d'accès dans ce fichier, de plus cette méthode entraînait la création d'un fichier par appel à cette fonction et cela pourrait poser des problèmes au niveau de l'OS [4] pour le gestionnaire de fichier. Le fichier créé va donc dépendre d'une instance de *Simul()* et nous pourront donc limiter le nombre de fichier en écriture, un seul fichier sera nécessaire pour simuler l'accès en écriture d'une BD, un seul par instance.

La librairie statique ***Tools*** reprend les méthodes suivantes :

- Une méthode ***proba()*** : cette méthode permet d'obtenir une valeure (booléenne) représntant un tirage au sort avec une probabilité donnée (par le paramètre de la méthode)
- Une méthode ***cpuInt()*** : cette méthode permet de simuler une charge processeur, il s'agit d'une boucle et chaque itération fait un calcul sur des entiers
- Une méthode ***cpuFloat()*** : cette méthode permet de simuler une charge processeur, il s'agit d'une boucle et chaque itération fait un calcul sur des réels
- Une méthode ***cpuString()*** : cette méthode permet de simuler une charge processeur, il s'agit d'une boucle et chaque itération fait un calcul une chaîne de caractères
- Une méthode ***createString()*** : cette méthode permet de créer une chaîne de caractère, la longueur est donné par le paramètre de la méthode

5.6 Le système de mesures

Le système de mesures proposé est un système très simple de classes qui héritent d'une super-classe (*Measure*).
Les mesures proposées sont les suivantes :

- ***Chrono*** : il s'agit d'une mesure de temps (en ms) faite entre 2 moments , il ne s'agit pas d'un temps CPU mais bien d'une durée entre 2 points de mesures
- ***Counter*** : il s'agit simplement d'un compteur que l'on incrémente à chaque appel
- ***Memory*** : il s'agit d'une mesure de la mémoire (en Bytes) utilisée à un instant T, il s'agit de la mémoire utilisée dans la JVM et non pas la mémoire au niveau du système d'opérations
- ***Cpu*** : il s'agit d'une mesure de charge CPU (en %) à un instant T

4. Operating System

La mesure de type Cpu pose cependant problème car aucune API Java ne permet de faire directement ce genre de mesure. Il existe des solutions mais nous n'avons pas réussi à les mettre en oeuvre étant données leurs complexités, nous avons cependant décidé de laisser la classe Cpu dans le framework pour une implémentation à postériori si une solution native devait voire le jour en Java.

Le diagramme de classe du système de mesures se trouve à la figure 5.4.

Figure 5.4: *Diagramme de classe du système de mesures*

5.7 Les fichiers résultats

Comme nous venons de le préciser ci-avant, les mesures sont des objets et il reste à sauvegarder ceux-ci afin de pouvoir en exploiter les résultats.
Nous devons donc définir un type de fichier ainsi que son format, celui-ci sera déterminant pour son exploitation. L'idéal pour exploiter les résultats est de pouvoir générer des courbes de valeurs, le permier logiciel qui nous vient à l'esprit pour créer des courbes facilement est bien évidemment *Microsoft Excel* mais rien n'empêche d'utiliser un autre, du moment que celui-ci accèpte le format définit des fichiers résultats.

Un des formats de fichiers les plus répendus et le plus facilement portable est le format *XML*[5], nous pouvons également associer à ce type de fichier des feuilles de styles, celles-ci permettent alors la génération de documents qui contiendront nos courbes de résultats. Ceci est le cadre idéal mais vu le contexte nous avons opté pour des fichiers au format *CSV*[6], et Excel permet justement d'importer des fichiers CSV directement dans ses feuilles.
Le format CSV est un format informatique ouvert représentant des données tabulaires sous forme de valeurs séparées par des virgules. Le fait que les fichiers CSV soient essentiellement utilisés autour du logiciel Microsoft Excel, et que les séparateurs ne soient pas standardisés (virgules, points-virgules (sous certaines localisations dont la française), etc.) rend ce format peu pratique pour une utilisation autre que des échanges de données ponctuelles. Ce format est toutefois assez populaire parce qu'il est relativement facile à générer et moyennant quelques manipulations manuelles, il sera assez facile de créer des courbes sur bases des mesures s'y trouvant.
Le format CSV présente entre autres désavantages d'être interprété par Excel, comme étant au format anglais (séparateur de colonnes : virgule ; séparateur de décimales : point) ou français de France (séparateur de colonnes : point-virgule ; séparateur de décimales : virgule) en fonction de l'origine du fichier : sur disque, par HTTP, support amovible, etc.. La figure 5.5 montre un exemple de fichier résultat.

On y trouve des champs séparés par le signe " ;" , dont la signification est la suivante :

1. Le nom de la machine
2. Le nom du composant
3. Le nom de la mesure
4. Le type de mesure (Chrono, Counter, Memory, ...)
5. L'unité de la mesure (ms, bytes, ...)
6. La valeur de la mesure

Quant au nom du fichier résultat, la nomenclature est la suivante :

`Result_File_<JJ>_<MM>_<AA>_<hh>_<mm>_<ss>_<msec>.csv`

- JJ : numéro du jour dans le mois [01-31]
- MM : numéro du mois dans l'année [1-12]
- AA : année sur 2 digits
- hh : heure [0-23]
- mm : minute [0-59]
- ss : seconde [0-59]
- msec : milliseconde [0-999]

5. XML : eXtensibled Markup Langage
6. CSV : Comma Separator Values

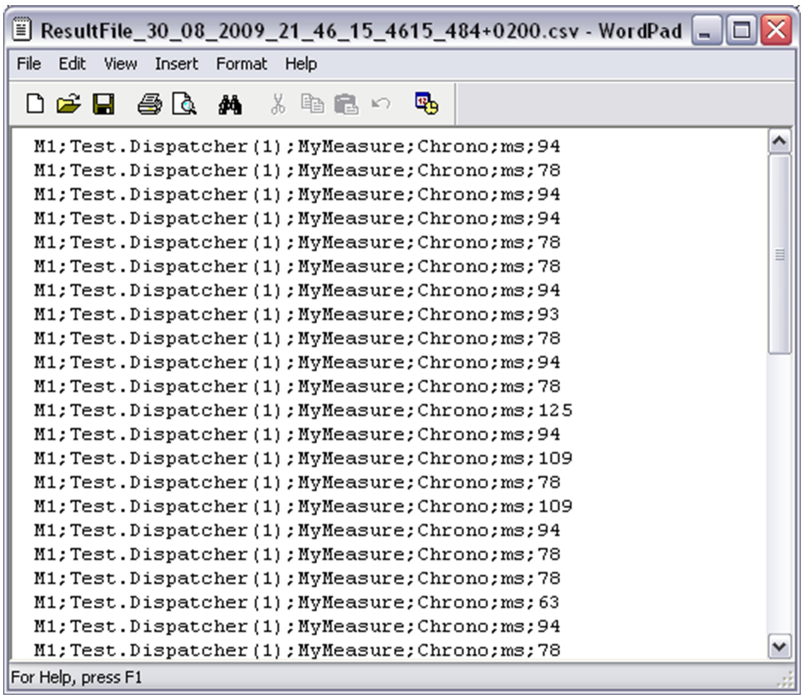

Figure 5.5: *Exemple de fichier résultat*

Les fichiers résultats sont sauvegardés dans le répertoire pointé par la propriété système *"'used.dir"'*+"'//ResultFile//"' comme sous-répertoire.
Nous pouvons ouvrir ces fichiers résultats avec un tableur et utiliser la colonne des valeurs pour créer des courbes comme dans la figure 5.6.

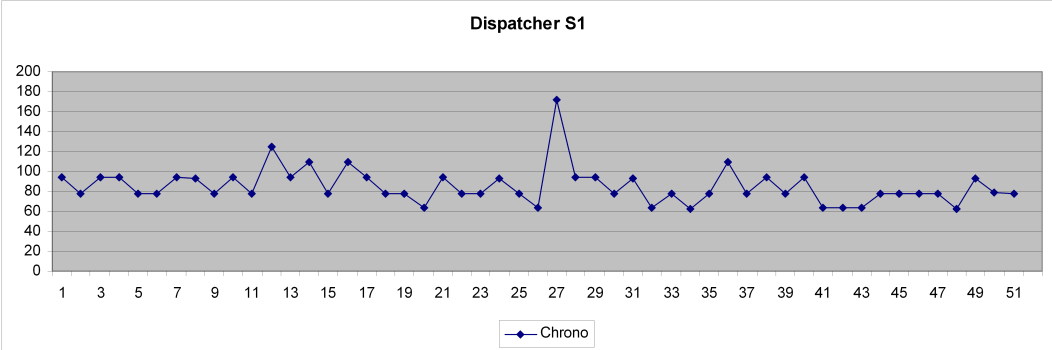

Figure 5.6: *Exemple de courbe générée dans Excel*

CHAPITRE 6

Conception et implémentation

6.1 Choix de l'implémentation

Nous avons vu dans le chapitre 3 que nous avons choisi Java/RMI pour sa facilité et simplicité de mise en oeuvre, mais il est important d'en dire un peu plus dans ce chapitre.

Le langage Java java.sun.com à été choisi pour implémenter le framework mais celui n'est pas forcément le mieux adapté à nos besoins.
Un des désavantages du langage Java est la lenteur d'exécution mais la performance en terme de temps d'exécution n'a pas d'importance ici. En effet, si on fait des comparaisons de temps d'appels de méthodes distantes entre composants, ceux-ci seront succeptibles de donner la même impression que si ces mesures sont faites dans un langage compilé comme par exemple le C ou C++, si c'est lent cela l'est pour tous les composants.
Un autre désavantage, dont nous avons vu les conséquences en cours d'élaboration de ce travail, est le *ramasse miette distant* (Remote Garbage Collector). Celui-ci libère la mémoire d'objets distants qui ne sont plus référencés (même fonctionnement que le Garbage Collector mais en distribué), le gros problème est que les objets "Machine", "Component" sont mémorisés dans des collections d'objets et tant que le système tourne Java ne sait pas libérer la mémoire car les objets distants sont toujours référencés via ces collections. Ce problème n'est cependant pas si pénalisant que cela car les PC d'aujourd'hui ont suffisament de mémoire pour pouvoir faire fonctionner notre application pendant un temp donné, à charge au développeur de scénarios d'arrêter le système puis de le relancer.

Le choix du langage présente cependant quelques avantages. Le premier est le fait que le langage est portable, c'est-à-dire que le même code Java peut-être exécuté sans modification sur diverses plate-formes.
Le langage Java est aussi un langage orienté objet et nous avons fait l'hypothèse que nous développons dans un paradigme orienté objet.
Le langage Java est particulièrement bien intégré à l'IDE Eclipse et il ne faut pas non plus négligé le fait que le JDK est fourni avec beaucoup de paquetages et librairies disponibles publiquement, sans devoir acquérir de licence particulière.

Nous avons aussi utilisé Java parque nous avons choisi RMI comme moyen de communication entre objets distants. Avoir choisi RMI est un choix de simplicité et aussi de connaissance, en effet nous avons étudié RMI lors des cours de *Concepts des Systèmes Coopératifs* et celui-ci rend facile la compréhension d'un code Java distribué alors que ce n'est pas forcément le cas avec CORBA et ses interfaces IDL.

6.2 Présentation du framework

Le framework est composé de plusieurs packages dont la découpe est la suivante :

- Un package **CO** : qui reprend les classes utiles côté chef d'orchestre à savoir :
  - La classe *CO* : c'est la classe principale
  - La classe *ScenarioPlayer* : il s'agit de la clase qui instancie le scénario par "reflection"
  - La classe *Choreography* : elle permet d'exécuter les demandes de création de machines, de composants, se trouvant dans le scénario
  - La classe *StartOneway* : cette classe permet d'appeler la méthode start() d'un composant de manière asynchrone, c'est-à-dire sans attendre le retour d'un résultat
- Un package **Collections** : qui reprend toutes les collections gérées par le framework :
  - La classe *Components* : il s'agit d'une collection de composants (Component) du chef d'orchestre
  - La classe *Machines* : il s'agit d'une collection de machine (Machine)
  - La classe *MachineComponents* : il s'agit d'une collection de composants (Component) d'une machine
- Un package **Component** : qui reprend toutes les classes utiles aux composants :
  - L'interface *Component* : il s'agit de l'interface distante qu'un composant doit implémenter
  - La classe *Parameters* : il s'agit de la classe décrivant les paramètres que l'on passe aux composants
  - L'interface *Params* : il s'agit de l'interface qu'un paramètre passé aux objets

distants doit implémenter
  - La classe *GenericComponent* : il s'agit de la super-classe d'un composant, elle implémente l'interface *Component*
- Un package **Factory** : qui reprend toutes les classes utiles à la fabrique de composant :
  - La classe *RMILauncher* : c'est le programme principale à démarrer sur les différentes machines du système
  - La classe *MachineImpl* : il s'agit de la classe décrivant une machine distante, l'instance de cette classe est enregistrée dans le RMIRegistry
  - L'interface *Machine* : il s'agit de l'interface de la classe MachineImpl
- Un package **Measure** : qui reprend toutes les classes utiles pour faire des mesures :
  - La classe *Measure* : c'est la super-classe des mesures
  - La classe *Chrono* : il s'agit de la classe décrivant une mesure de type chronomètre
  - La classe *Counter* : il s'agit de la classe décrivant une mesure de type compteur
  - La classe *Quantity* : il s'agit de la super-classe décrivant une mesure de type quantité
  - La classe *Memory* : il s'agit de la classe décrivant une mesure de type mémoire utilisée
  - La classe *Cpu* : il s'agit de la classe décrivant une mesure de type Cpu

6.2.1 Le chef d'orchestre

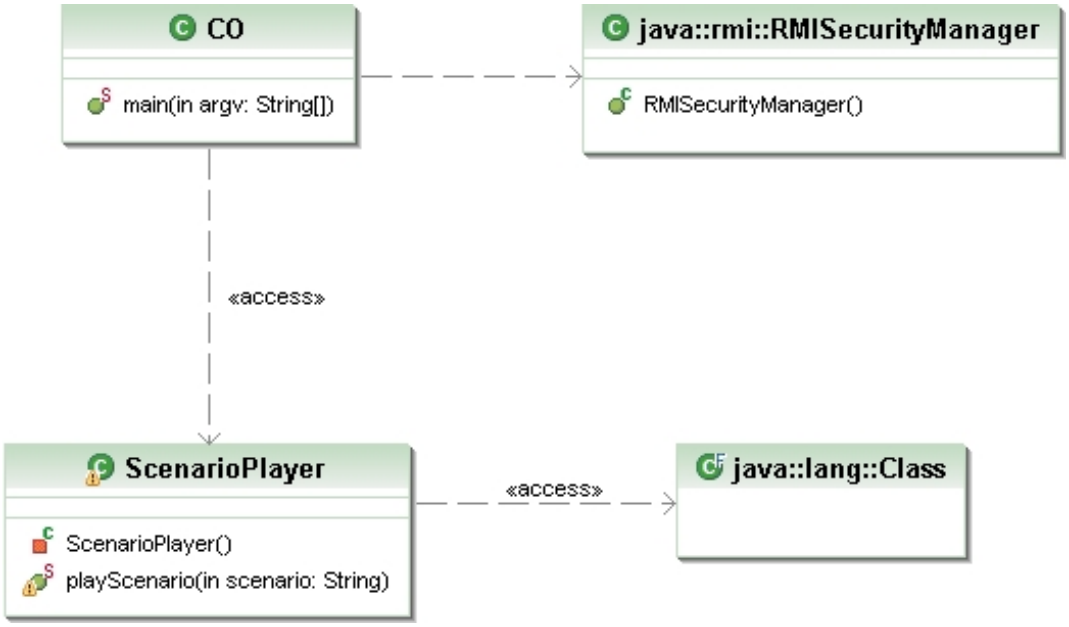

Figure 6.1: *Le diagramme de classes du CO*

Dans le diagramme de classe du CO nous pouvons voir que la classe CO est le programme principal à démarrer côté chef d'orchestre. Ce programme prend en argument un nom de scénario à exécuter, le CO va appeler la classe ScenarioPlayer() en lui passant le nom du scénario et celle-ci va se charger d'instancier le scénario via l'API Relfection.

Le CO crée également une nouvelle instance de RMISecurityManager afin de contrôler les accès aux méthodes distantes. Pour ce faire dans les paramètres de la compilation Javac, nous devons préciser quel est le fichier qui définit la policy, dans notre framework ce fichier se nomme *security.policy* et son contenu est le suivant :

```
// This file was generated by the RMI Plugin for Eclipse.

////////////////////////////////////////////////////////////
// This is a sample policy file that grants the application all permissions.
// A policy file is needed by the RMISecurityManager and your application
    might
// not work after installing the RMISecurityManager unless you provide a
// security policy file at launch.
//
// You can configure the security policy of a launched application using
    either
```

```
// the RMI Launcher or by manually setting the java.security.policy property.
//
// SECURITY NOTE: This security policy is good for development. For deployment
// you may need a stricter security policy.
//
// For more information see:
//    http://java.sun.com/docs/books/tutorial/rmi/running.html
//    http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html
//

grant {
        permission java.security.AllPermission;

        // Other options:
        // permission java.net.SocketPermission "127.0.0.1:1024-", "accept,
            connect, listen, resolve";
        // permission java.net.SocketPermission "localhost:1024-", "accept,
            connect, listen, resolve";

        // From http://java.sun.com/docs/books/tutorial/rmi/running.html
        // Copyright 1995-2005 Sun Microsystems, Inc. Reprinted with
            permission

        // permission java.net.SocketPermission "*:1024-65535", "connect,
            accept";
        // permission java.net.SocketPermission "*:80", "connect";

        // permission java.net.SocketPermission "*:1024-65535", "connect,
            accept";
        // permission java.io.FilePermission "c:\\home\\ann\\public_html\\
            classes\\-", "read";
        // permission java.io.FilePermission "c:\\home\\jones\\public_html\\
            classes\\-", "read";
};
```

Certe donner toutes les permissions n'est pas à conseiller, mais dans le cadre de ce mémoire cela n'est pas contraignant, bien au contraire car il faudrait modifier souvent ce fichier si de nouvelles machines devaient-être prises en compte dans le système.

6.2.2 Les collections

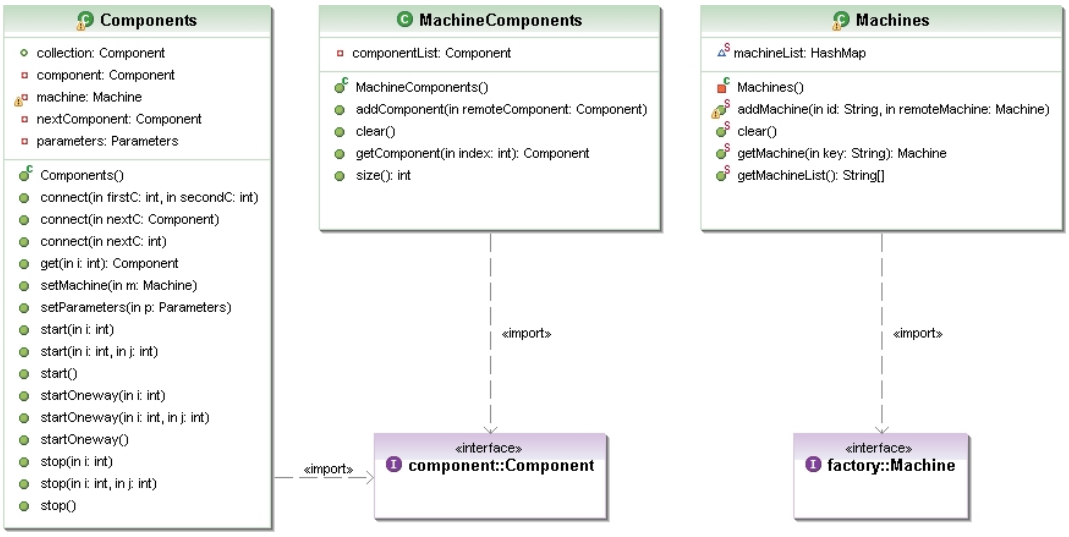

Figure 6.2: *Le diagramme de classes des collections*

Dans le diagramme de classes des collections, nous retrouvons une classe très importante côté chef d'orchestre. En effet la classe *Components()* est une collection de composants (*Component*) qui va se charger d'appeler une méthode distante sur des objets de sa collection, cette classe sert en fait d'appel local avant de faire un appel de méthode sur un composant distant.

Etant donné qu'il s'agit d'une collection qui est créée dans le scénario, plutôt que d'appeler une méthode sur chaque composant distant et de devoir gérer les exceptions au niveau du scénario, on va appeler une méthode sur la collection de composants du CO. Les méthodes disponibles de la collection ne font qu'appeler la méthode souhaitée sur chaque élément de la collection.

Voici les principales méthodes disponibles de la collection *Components()* :

- *connect(Component nextC)* : demande à tous les composants de la collection de se connecter au composant **nextC**
- *connect(int nextC)* : demande à tous les composants de la collection de se connecter au composant numéro **nextC**, le premier composant de la collection a l'indice 1
- *connect(int firstC, int secondC)* : demande au composant firstC de se connecter au composant secondC via leur position dans la collection
- *start()* : appelle la méthode start() de chaque composant de la collection (en synchrone)

- *start(int i)* : appelle la méthode start() du composant i de la collection
- *start(int i, int j)* : appelle la méthode start() du composant i à j de la collection
- *startOneway()* : appelle la méthode start() de chaque composant de la collection (en asynchrone) via la classe *StartOneway()*
- *startOneway(int i)* : appelle la méthode start() en asynchrone du composant i de la collection
- *startOneway(int i, int j)* : appelle la méthode start() en asynchrone du composant i à j de la collection
- *stop()* : appelle la méthode stop() de chaque composant de la collection (en synchrone), la constante RUNNING sera positionnée à False et celle-ci pourra éventuellement être utilisée pour arrêter une boucle while par exemple
- *stop(int i)* : appelle la méthode stop() du composant i de la collection
- *stop(int i, int j)* : appelle la méthode stop() du composant i à j de la collection

Quant aux collections Machines() et MachineComponents(), il s'agit tout simplement d'une collection de *Machine()* et d'une collection de composants mais utilisée par les Machines, pour par exemple pouvoir appeler la méthode *stop()* de chaque composant d'une machine.
Les méthodes disponibles ne demande pas une grande explication, on peut ajouter un objet dans la collection avec *add()*, demander une référence d'objet avec *get()* ou également donner la taille de la collection et aussi pouvoir effacer la collection via la méthode *clear()*.

6.2.3 Les classes utiles aux composants

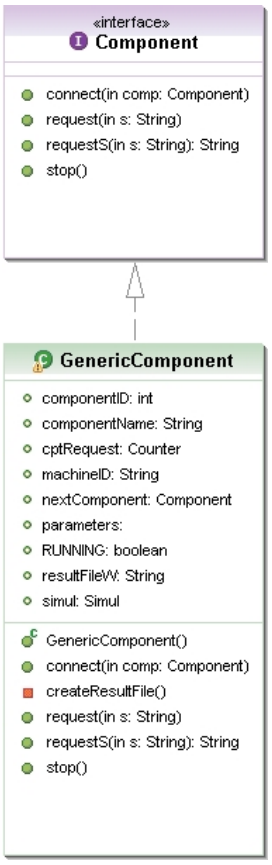

Figure 6.3: *Le diagramme de classes des composants*

Un des aspects important dans notre framework est que les objets qui doivent communiquer entre-eux via les appels de méthodes distantes sont considérés comme étant des composants (*Component*). Nous avons donc créé une interface distante que tous les composants du système doivent implémenter, c'est-à-dire que toutes les méthodes déclarées dans l'interface doivent trouver leur implémentation dans la classe qui implémente cette interface.
Afin de regrouper certaines informations (les attributs) et certaines fonctions communes entre les différents composants, une super-classe de composant à été créé (*GenericComponent()*) et c'est elle qui implémentera l'interface *Component()*.
Avec les méthodes déclarées au niveau de l'interface *Component()*, tous les scénarios envisagés et toutes les stratégies possibles que l'on a étudiés sont couverts par ces seules méthodes.
Voici les principales méthodes disponibles de l'interface *Component()* :

- *connect(Component nextC)* : mémorise dans l'attribut *nextComponent* la référence l'objet distant à appeler
- *start(Params p)* : méthode qui est appelée par le scénario et reçoit en paramètre la table de hachage, que l'on pourra ou non utiliser selon la politique choisie dans la dynamique du système
- *void request(String s)* : méthode qui reçoit en paramètre une chaîne de caractères mais ne retourne rien
- *String requestS(String s)* : méthode qui reçoit en paramètre une chaîne de caractères et qui retourne une chaîne de caractères
- *stop()* : méthode qui est appelée par le scénario ou par la fabrique pour arrêter un composant, via sa constante RUNNING

Quant aux composants que nous devons écrire pour simuler un prototype de SD, ils devront hériter de la classe *GenericComponent()* et par exemple les méthodes *start()*, *request()* et *requestS()* devront-être surchargées afin de personaliser un comportement d'un composant voir le chapitre 7 pour des exemples de composants.
La classe *GenericComponent()* possède une méthode privée *createResultFile()*, celle-ci est appelée par le constructeur de la classe afin de créer une nom de fichier à utiliser pour la sauvegarde des résultats.

6.2.4 La fabrique de composants

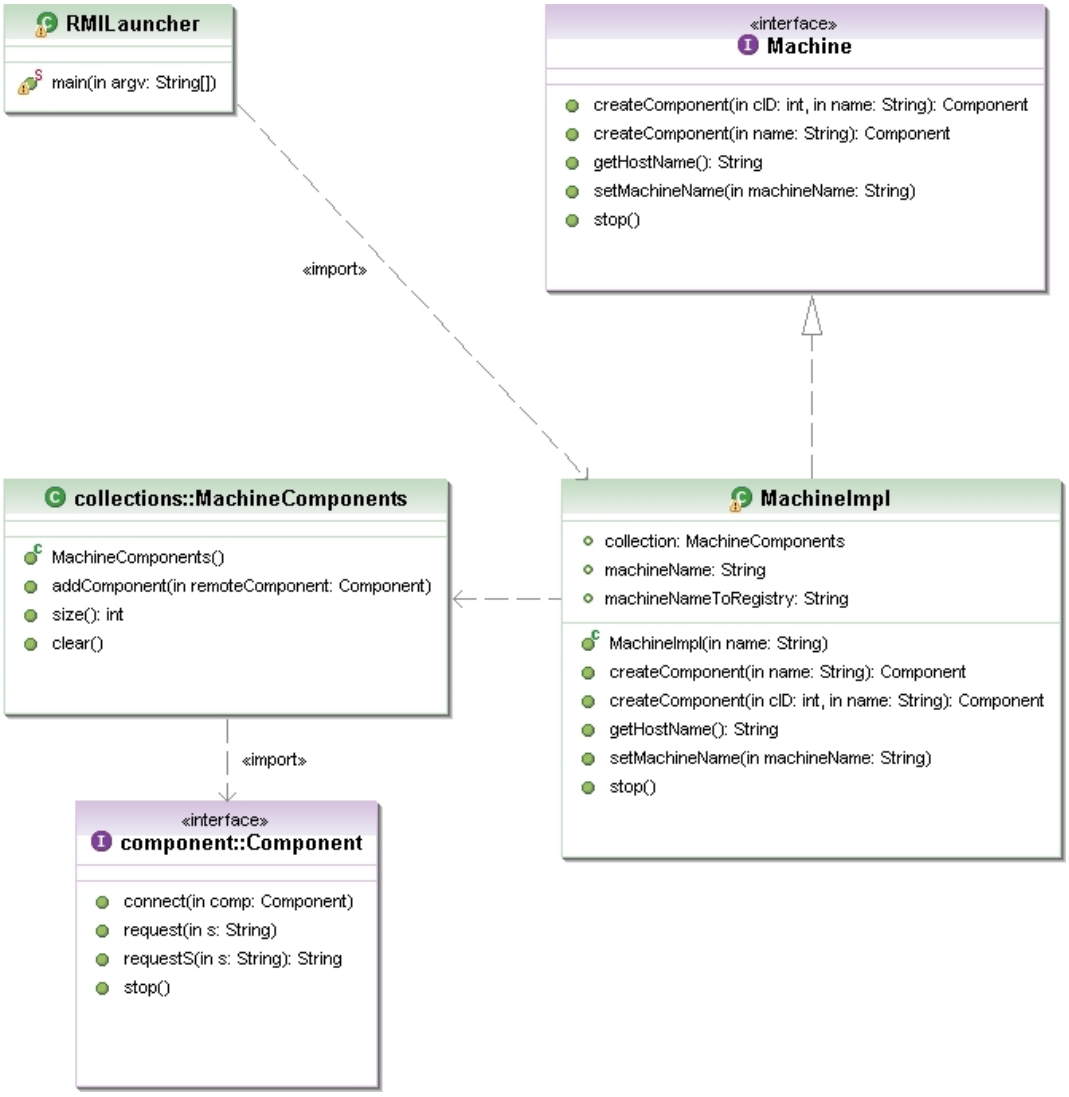

Figure 6.4: *Le diagramme de classes de la fabrique*

La fabrique de composants est, comme nous l'avons dit au début de ce mémoire, un programme qu'il faut démarrer sur chaque machine faisant partie du système distribué. Le programme à démarrer sur chaque PC est *RMILauncher*, comme le CO il va créer un RMISecurityManager() mais aussi un RMIRegistry() sur le port 1099 (port par défaut). Le programme va instancier la classe *MachineImpl()* en lui passant comme paramètre le nom à publier dans le registre, dans notre cas il s'agit

de "'Machine"'.

La classe MachineImpl() va publier sa référence dans l'annuaire et lui associera le mot "'Machine"' comme mot clé. Dès cette instant, la fabrique d'objets est opérationnelle et ses méthodes sont disponibles grâce à l'interface *Machine()* que la classe implémente.
Les principales méthodes distantes disponibles pour la fabrique sont les suivantes :

- *Component createComponent(String name)* : crée un objet specifié par *name*, l'instanciation se fait par Reflection et en lui ajoutant le mot "'Impl"' pour le nom de la classe Java à instancier, les attributs du composant sont mis à jour également par Reflection (nom du composant, identifiant, etc.) et l'objet ainsi crée est ajouter à la collection de composants de la Machine, la méthode retourne la réference de l'objet distant que la fabrique vient de créer
- *stop()* : appelle la méthode *stop()* de chaque composant de la collection de composants de la machine et ensuite efface la collection

6.2.5 La dynamique du système

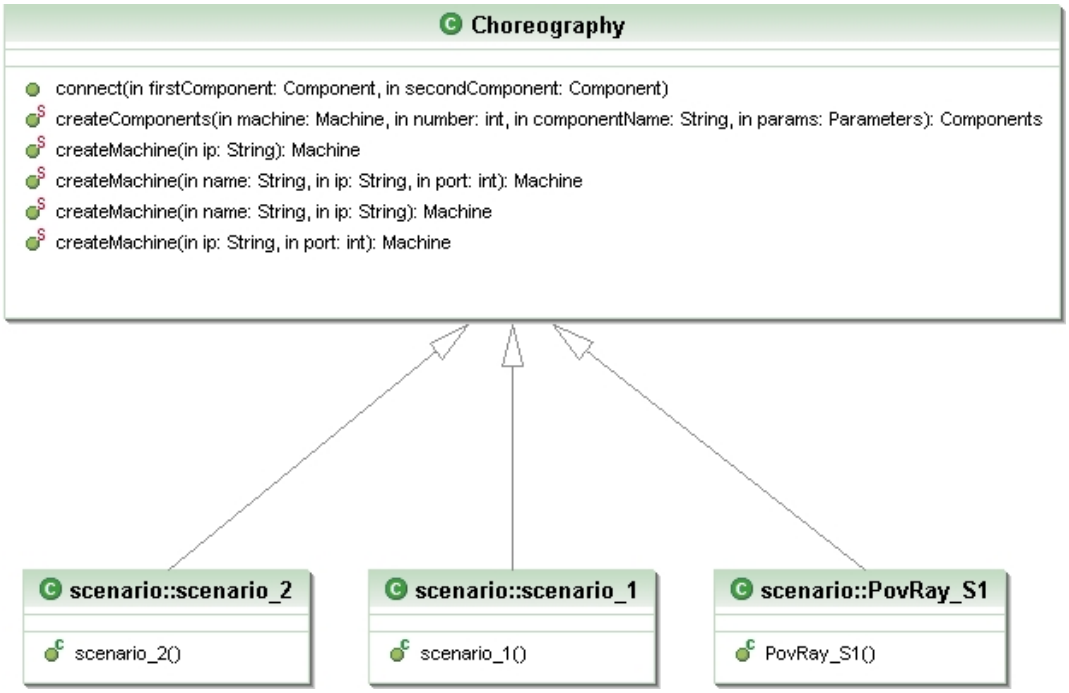

Figure 6.5: *La classe choreography et ses dépendances*

Dans le diagramme de classes de la chorégraphie, on voit que chaque scénario que l'on écrit doit hériter de la classe *Choreography()*, les principales méthodes disponibles sont les suivantes :

- *Machine createMachine(String ip, int port)* : permet d'obtenir la référence de la fabrique se trouvant sur le PC dont l'adresse ip et le port sont précisés via les paramètres de la méthode
- *Machine createMachine(String ip)* : permet d'obtenir la référence de la fabrique se trouvant sur le PC dont l'adresse ip (port par défaut étant le 1099) est précisée via le paramètre de la méthode
- *Machine createMachine(String name, String ip, int port)* : permet d'obtenir la référence de la fabrique se trouvant sur le PC dont l'adresse ip et le port sont précisés via les paramètres de la méthode et de plus on peut spécifié la clé à utiliser pour ajouter cette référence dans la collection de Machines
- *Machine createMachine(String name, String ip)* : idem que la précédente mais le port est le 1099 par défaut
- *Components createComponents(M, N, C, P)* : permet de demander à une machine M de créer N composants C en leur passant le paramètre P, et la méthode retourne une collection de composants (Components)
- *connect(Component firstComponent, Component secondComponent)* : prévient le composant *firstComponent* qu'il doit se connecter au composant *secondComponent*

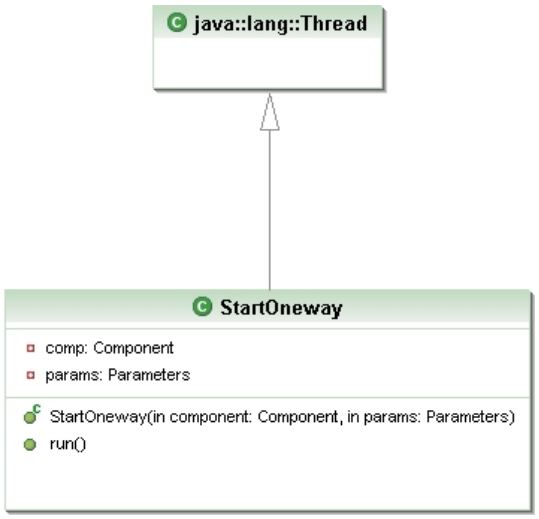

Figure 6.6: *La classe startOneway pour un appel asynchrone*

Dans la figure 6.6, on voit que la classe StartOneway() hérite de la classe Thread, cela veut dire que la méthode start() d'un composant sera appelée de manière asyn-

chrone donc sans attendre de retour de résultat. On utilisera cette méthode surtout pour les méthodes qui contiennent des boucles et qui effectivement doivent-être appelées de manière asynchrone sous peine de bloquer l'appelant.

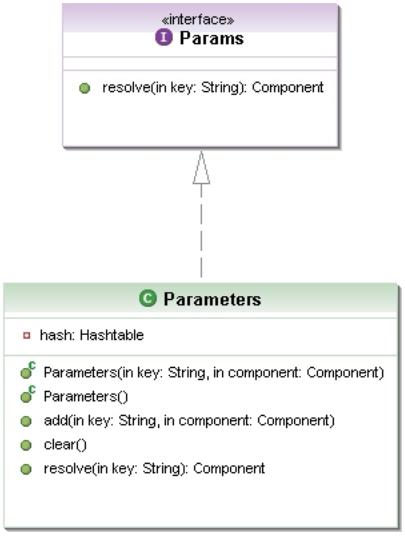

Figure 6.7: *La classe Parameters et son interface*

Nous voyons dans la figure 6.7 qu'avec cette classe et son interface, nous pouvons passer un objet de type *Params*, qui est en fait une table de hachage, aux objets distants.

6.2.6 Les mesures

Figure 6.8: *Le diagramme de classe du système de mesures*

Le système de mesures proposé à la figure 6.8 est un système simple de classes qui héritent d'une super-classe. Les méthodes des classes dérivées peuvent surcharger celles de la classe *Measure()*. La raison à cela est qu'une mesure de type chronomètre donne un temps exprimé en millisecondes tandis qu'une mesure de type compteur ne fait qu'incrémenter une valeur entière, donc les méthodes *start()* et *stop()* seront différentes en fonction du type de mesure.
Tous les attributs communs ont été regroupés dans la super-classe *Measure()*, à savoir :

- *long value* : valeur de la mesure exprimée en réelle long
- *String unit* : unité de la mesure, celle-ci est précisée dans les classes filles et sera sauvegardée dans le fichier résultat
- *boolean terminated* : flag indiquant si la mesure est terminée, utile pour sauver la mesure
- *String measureName* : nom de la mesure, sera sauvegardée dans le fichier résultat
- *String componentName* : nom du composant concerné par la mesure

- *String measureType* : type de mesure, sera sauvegardé dans le fichier résultat
- *String resultFileW* : nom du fichier résultat ou les mesures seront sauvegardées

6.3 Implémentation d'un problème

Nous voyons dans cette section ce qu'il faut faire pour implémenter un problème avec notre framework, voir le chapitre 5 pour plus de détails et le chapitre 7 pour plus d'exemples.

Pour implémenter un problème, la première chose à faire est de définir les types de composants que l'on doit implémenter. Ensuite il faut définir ce que chaque composant doit faire, que veut-on simuler comme charge pour que cela donne une impression par rapport au système réel que l'on veut émuler.

Il faut décrire le scénario que l'on veut exécuter, à savoir :

1. Définir le nombre de machines faisant partie du système
2. Définir le type et le nombre de composants à créer sur une machine donnée
3. Définir la méthode à utiliser pour la dynamique du système, cela peut dépendre du scénario, on peut mélanger les deux, (cf. l'étude de cas *PovRay*)
4. Préciser justement quel composant se connecte à quel composant
5. Donner l'ordre et le type de démarrage au niveau des composants, synchrone avec la méthode *start()* ou asynchrone avec la méthode *startOneway()*
6. Eventuellement faire une pause (via la méthode *pause(x msec)*) avant d'arrêter le système via la méthode stop() des composants ou des machines.

Chaque classe qui définit un composant doit étendre (hériter de) la classe *GenericComponent()*, celle-ci possède des méthodes qui permettent de faire déjà pas mal de choses, ces méthodes sont les suivantes :

- *start(Params)* : cette méthode est toujours à appeler dans le scénario, elle initie le composant et ses paramètres, mais elle peut aussi contenir des boucles avec plusieurs éléments de simulation de charge ou autre
- *stop()* : cette méthode positionne la constante RUNNING a False, ce qui permettra (selon le cas) l'arrêt du composant dans le cas par exemple ou la méthode *start()* serait une boucle qui dépend de la constante (cf. les études de cas)
- *connect(Component)* : cette méthode précise quel sera le prochain objet distant à appeler dans le chaînage du système
- *request(String)* : cette méthode reçoit en argument une chaîne de caractères et ne retourne rien, l'appelant de doit donc pas s'attendre à recevoir une réponse (utile pour des appels asynchrones)
- *requestS(String)* : cette méthode reçoit en argument une chaîne de caractères et retourne une chaîne de caractères (utile pour des appels synchrones)

CHAPITRE 7

Etude de cas

7.1 Introduction

L'élaboration de ce mémoire est basé sur le prototypage d'architectures distribuées. Il est donc question d'étudier quelques cas de SD afin d'en abstraire le principe de fonctionnement et d'en faire une simulation. Pour créer le framework et ses besoins, nous sommes partis de la description d'un système distribué et nous en avons extrait le principe de fonctionnement afin d'en faire une simulation.

Les systèmes étudiés pour ce travail sont :

1. Un simple Client - Serveur (un serveur de page web)
2. Un système P2P (le système Gnutella)
3. Le TP MDL sur le rendu d'image 3D povray

Dans la simulation d'un prototype de SD, plusieurs stratégies peuvent-être implémentées, nous ne pouvions pas toutes les tester mais nous nous sommes assurer que les trois études de cas fonctionnent avec les concepts mis en place. Il faut noté également que les résultats de tests inclus dans ce document pour les systèmes décrits ci-après proviennent d'un seul ordinateur, la raison à cela est que la sauvegarde des mesures dans des fichiers résultats n'à été disponible que tout à la fin de ce travail et nous ne disposions plus que d'un seul ordinateur pour faire fonctionner le système. Néanmoins les trois études de cas ont été également valider sur un réseau constitué de plusieurs PC afin de valider le framework, la seule différence au niveau des résultats était que les temps d'appels de méthodes distantes étaient un peu plus long.

7.2 Le simple Client-Serveur

7.2.1 Description du problème

Le premier système étudié pour évaluer les besoins en terme de simulation est un simple serveur de page web, simple parce qu'il n'y a que peu d'interactions entre les composants, mais celui-ci pose déjà un bon nombre de questions.

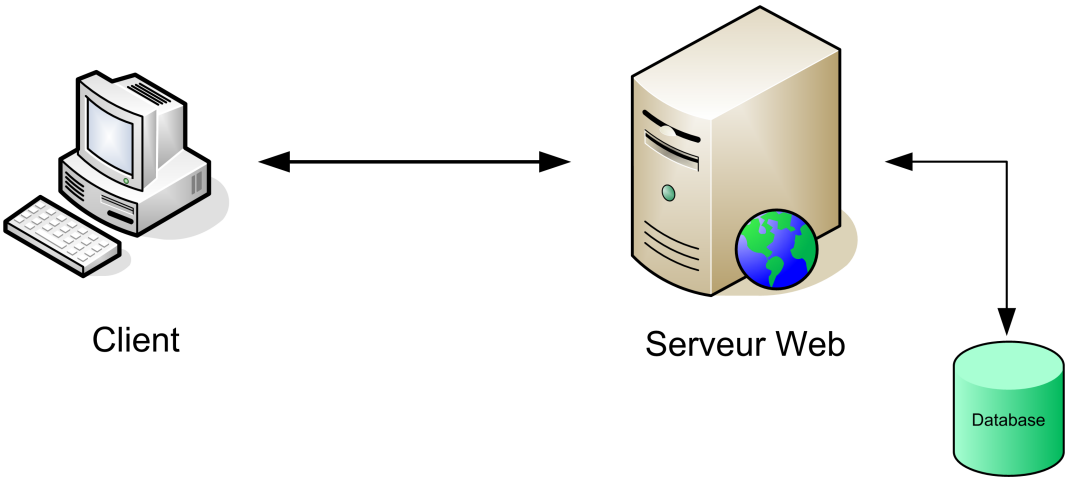

Figure 7.1: *Serveur de page web*

Un client se connecte à un serveur afin de télécharger une page web, lorsque le serveur est sollicité il recherche la page soit dans des fichiers soit dans une base de données et la retourne au client, celui-ci peut dès lors afficher son contenu.

Dans notre simulation ce qui nous intéresse c'est de connaître le temps que prend l'appel au serveur du côté du client, et côté serveur ce qui peux-être interressant c'est de connaître le nombre d'invocation de la méthode qui retourne les pages web.

Les questions que l'on peut se poser dans ce cas sont les suivantes :
On sait que nous avons un client qui se connecte à un serveur et chaque requête sur le serveur prend un certains temps.

1. Quel sera le temps par invocation si on a 100 clients qui font la même chose et en même temps ?
2. Le temps d'invocation d'une requête sur le serveur par client sera-t-il significativement différent ?

7.2.2 Implémentation des composants

D'un point de vue abstrait, l'architecture comporte deux types de composants :

- **Le client :** c'est un composant qui fait appel au serveur pour recevoir une page web et afficher son contenu.
- **Le serveur :** le serveur gère les requêtes des clients et retourne un contenu de page web aux demandeurs.

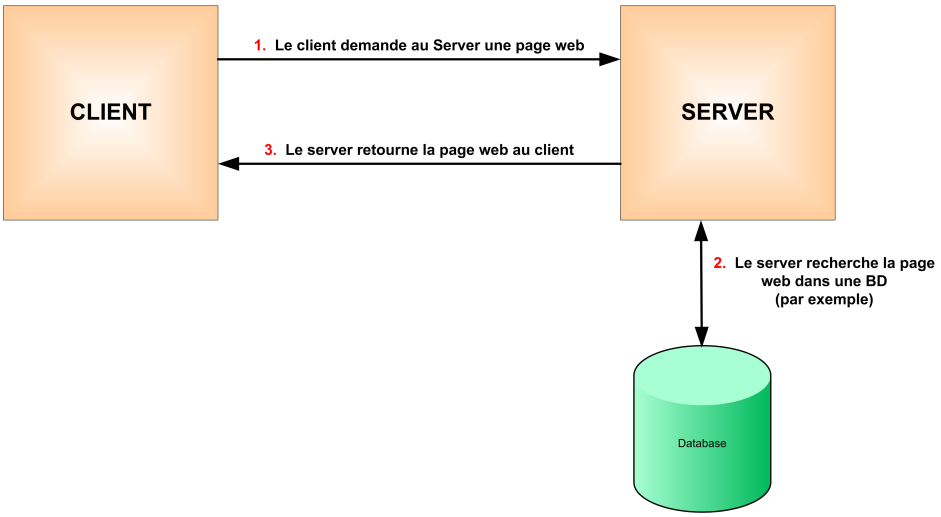

Figure 7.2: *Les composants du Client-Serveur*

Le scénario

```
package scenario;

import java.rmi.RemoteException;

import factory.Machine;
import collections.Components;
import co.Choreography;
import component.Parameters;
import static tools.Tools.*;

public class scenario_1 extends Choreography {

        public scenario_1() {

                Machine M1 = createMachine("M1", "127.0.0.1");

                // Creation de 1 composant de type "Serveur" sur la machine M1
                Components S = createComponents(M1, 1, "Case1_WebServer.
                    Serveur", null);

                // Creation de 1 composant de type "Client" sur la machine M1
                Components C = createComponents(M1, 1, "Case1_WebServer.Client
                    ", null);
```

```
                S.start();

                // SYSTEM DYNAMICS
                // All the C component must be connected to component "S.get
                    (1)"
                C.connect(S.get(1));
                C.startOneway();

                pause(30000); // Wait X ms.

                try {
                        M1.stop();
                } catch (RemoteException e) {
                        e.printStackTrace();
                }

        }

}
```

Le Client

```
package component;

import java.rmi.RemoteException;
import measure.Chrono;
import static tools.Tools.*;
import co.Params;

public class ClientImpl extends GenericComponent implements Client {

        public ClientImpl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);

                Chrono chrono = new Chrono(this,"MyMeasure");
                while (RUNNING) {
                        chrono.start();
                        if (proba(33)) {
                                nextComponent.request(createString(100));
                        } else {
                                nextComponent.request(createString(2000));
                        }
                        chrono.stop();
                        chrono.display();
                        chrono.save();

                        // Time Loop = 1 sec.
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }

                }
```

```
        }

        public void request(String s) throws RemoteException {
                super.request(s);
                super.cptRequest.display();
        }

}
```

Le Serveur

```
package component;

import static tools.Tools.createString;
import static tools.Tools.proba;

import java.rmi.RemoteException;

import measure.Chrono;
import measure.Counter;
import measure.Measure;

import static tools.Tools.*;

import co.Params;


/**
 * @author JC
 *
 */
public class ServeurImpl extends GenericComponent implements Serveur {


    public ServeurImpl() throws RemoteException {

    }

    public void start(Params p) throws RemoteException {
        super.start(p);
    }

    public synchronized void request(String s) throws RemoteException {
        super.request(s);

        if (proba(33)) {
                                cpuFloat(100);
                } else {
                        disk(1000,'w');
                }
        super.cptRequest.display();

    }
```

```
}
```

7.2.3 Résultats des tests

Nous avons d'abord fait fonctionner le système pendant 1 minute afin simplement de s'assurer que le résultat donnait une impression correcte par rapport au code des composants. Pour rappel, dans 1/3 des cas on simule une charge cpu avec *cpuFloat(100)* et dans les 2/3 des cas on simule l'accès à une BD *disk(1000,'w')*. Nous devrions retrouver 1/3 des mesures avec une certaine valeur et 2/3 des mesures avec une autre valeur.

Dans la figure 7.3 nous voyons que nous avons bien une valeur qui se présente 1 fois

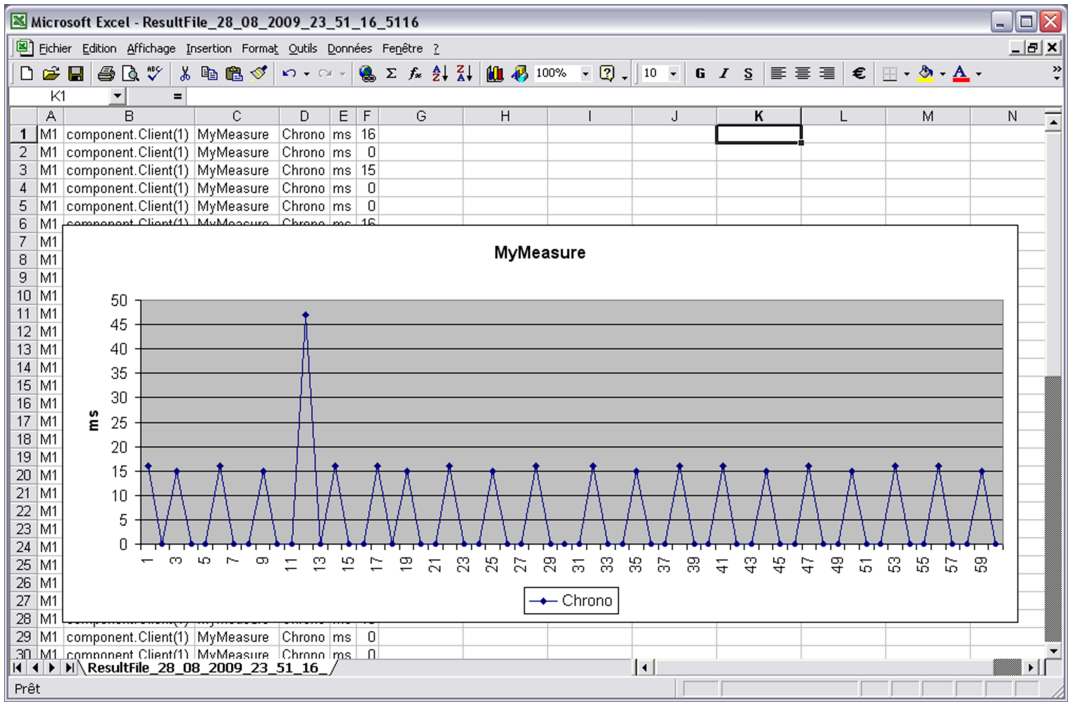

Figure 7.3: *Résultat pour un client et un serveur*

sur 3 et une autre valeur dans le reste des cas.

7.3 Le système P2P Gnutella

7.3.1 Description du problème

Reprenons le fonctionnement (simplifié) d'un système P2P comme Gnutella décrit auparavant. On a un programme qui va se déployer sur les 3 ordinateurs (M1,M2,M3) et si on définit le comportement du programme de manière très simple, ce programme va recevoir 1 requête et répondre (donc c'est synchrone).

- M1 reçoit une requête, il cherche localement dans sa BD s'il trouve quelque chose et il demande aux autres machines de faire la même chose
- Sur la figure 7.4 la requête arrive sur M1, elle se propage sur M2 puis sur M3, et une fois que l'on détecte la boucle (retour sur M1), le résultat retourne d'ou il vient

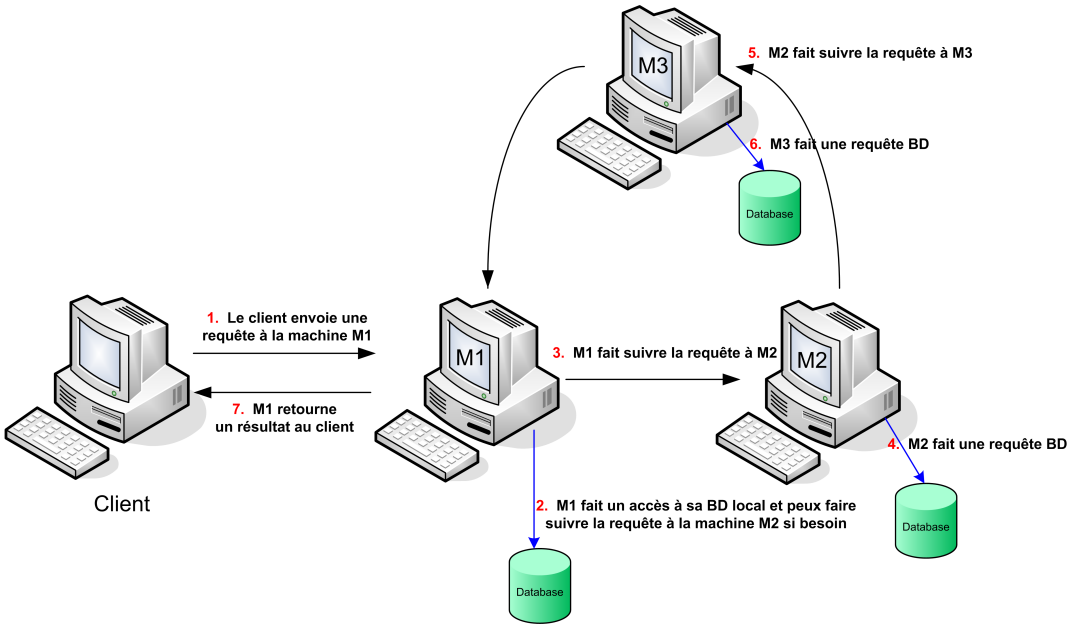

Figure 7.4: *L'architecture P2P Gnutella*

Si on veut programmer cela un peu à la louche, on obtient quelque chose comme ceci :

1. un service (une méthode) "'Request"' qui reçoit en argument une chaîne de caractères et qui retourne une autre chaîne de caractères, le contenu n'a aucune importance ici,
2. ce service doit accomplir une tâche bien précise à savoir :
   (a) simuler l'accès à la BD en faisant un accès à un fichier local

(b) faire suivre la requête au composant suivant sauf si c'est le premier dans ce cas on retourne une chaîne de caractères au client

7.3.2 Implémentation des composants

D'un point de vue abstrait, l'architecture comporte deux types de composants :

- **Client :** c'est un composant qui fait appel au premier composant de M1 et qui attend le retour d'information, il mesure le temps d'appel de la méthode distante de M1.
- **P2P :** ce composant gère les requêtes des clients et retourne une chaîne de caractères aux demandeurs, il propage également la requête aux autres composants P2P.

Le scénario

```
package scenario;

import java.rmi.RemoteException;

import factory.Machine;
import collections.Components;
import co.Choreography;
import component.Parameters;
import static tools.Tools.*;


public class scenario_2 extends Choreography {


        public scenario_2() {

                Machine MC = createMachine("MC", "127.0.0.1");
                //Machine M1 = createMachine("M1", "127.0.0.1");
                //Machine M2 = createMachine("M2", "127.0.0.1");
                //Machine M3 = createMachine("M3", "127.0.0.1");

                // Creation de 1 composant de type "ClientP2P" sur la machine
                    M1
                Components Client = createComponents(MC, 1, "Case2_P2P.Client"
                    , null);

                // Creation de 3 composant de type "P2P" sur la machine M1
                Components A = createComponents(MC, 3, "Case2_P2P.P2P", null);


                // SYSTEM DYNAMICS
                // All the C component must be connected to component "S.get
                    (1)"
                connect(Client.get(1),A.get(1));
                connect(A.get(1),A.get(2));
                connect(A.get(2),A.get(3));
                connect(A.get(3),A.get(1));
```

```
                A.start();
                Client.startOneway();

                pause(30000); // Wait X ms.
                // Stop the system
                try {
                        MC.stop();
                } catch (RemoteException e) {
                                e.printStackTrace();
                }


        }

}
```

Le Client

```
package Case2_P2P;

import java.rmi.RemoteException;
import measure.*;
import component.*;

import static tools.Tools.*;

public class ClientImpl extends GenericComponent implements Client {

        public Chrono chrono;
        public String requete;

        public ClientImpl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);

                chrono = new Chrono(this, "MyChrono");

                while (RUNNING) {

                        chrono.start();

                        System.out.println("Sending of request to P2P");
                        requete = nextComponent.requestS(createString(200));
                        System.out.println("Receiving of request from P2P");

                        chrono.stop();
                        chrono.display();
                        chrono.save();


                        // Time Loop = 1/2 sec.
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e) {
```

```
                                e.printStackTrace();
                        }

                }

        }

        public String requestS(String s) throws RemoteException {

                return s;
        }

        public void request(String s) throws RemoteException {

        }

}
```

Les Composants P2P

```
package Case2_P2P;

import java.rmi.RemoteException;
import measure.*;
import component.*;
import static tools.Tools.*;

public class P2PImpl extends GenericComponent implements P2P {


        public Chrono c1, c2;

        public P2PImpl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);

                c1 = new Chrono(this, "C1");
                c2 = new Chrono(this, "C2");
        }

        public String requestS(String s) throws RemoteException {
                // On déclenche le chrono
                c1.start();
                // On reçoit la requête
                // On fait une requête BD en lecture
                simul.disk(1000, 'r');

                // On passe au composant suivant la requête que l'on a reçu
                nextComponent.request(s);

                // On arrêt la mesure et on sauvegarde le résultat
                c1.stop();
                c1.display();
                c1.save();
```

```
                return createString (2000);
        }

        public void request(String s) throws RemoteException {

                // Si on est le 1er composant on ne fait plus le chaînage d'
                    objet
                String str = componentName.substring(componentName.indexOf("("
                    )+1, componentName.indexOf(")"));
                if ( str.equals("1") ) {
                        // nothing
                } else {
                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        c2.start();
                        // On reçoit la requête
                        // On fait une requête BD en lecture
                        simul.disk(1000, 'r');
                        // On passe au composant suivant la requête que l'on a
                             reçu
                        nextComponent.request(s);
                        c2.stop();
                        c2.display();
                        c2.save();
                }
        }
}
```

7.3.3 Résultats des tests

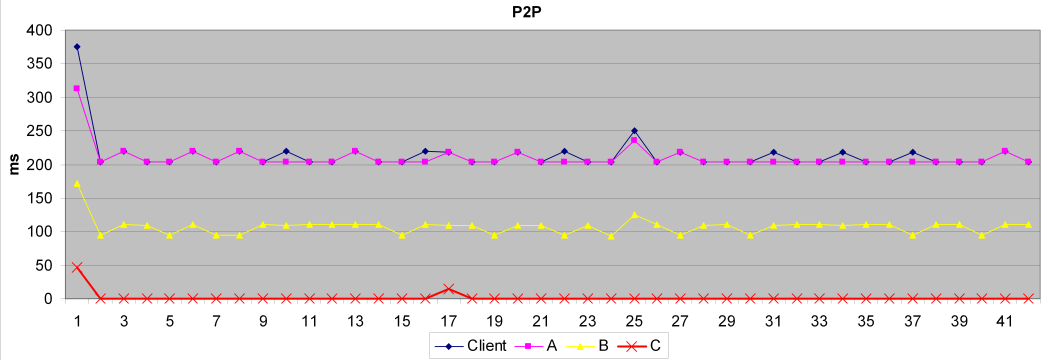

Figure 7.5: *Courbes des mesures du système P2P*

7.4 Le TP MDL de rendu d'images 3D povray

7.4.1 Description du problème

En 2005, le TP du cours de Conception des Systèmes Distribués demandait d'écrire une application distribuée qui permettrait d'accélérer le processus de rendu d'images en 3D comme le fait déjà *PovRay* (the Persistence of Vision Ray-Tracer), en appliquant le principe du *Grid Computing*[1].
Voici un extrait de l'énnoncé de ce TP :
"Le logiciel PovRay crée des images 3D en utilisant une technique de rendu appelée ray-tracing. Le logiciel lit un fichier texte contenant des informations décrivant les objets et les lumières dans une scène du point de vue d'une caméra décrite aussi dans une fichier texte. Le ray-tracing est un processus lent mais qui produit des images 3D de très bonne qualité.
Le but du travail était de rendre l'application PovRay distribuée car il est en effet facile d'imaginer que l'on pourrait fragmenter l'image dont il faut faire le rendu en plusieurs parties, de distribuer chacune de ces parties à des instances de PovRay résidant sur des machines différentes, de récupérer les images produites par ces instances, et de les rassembler pour reconstituer l'image complète".

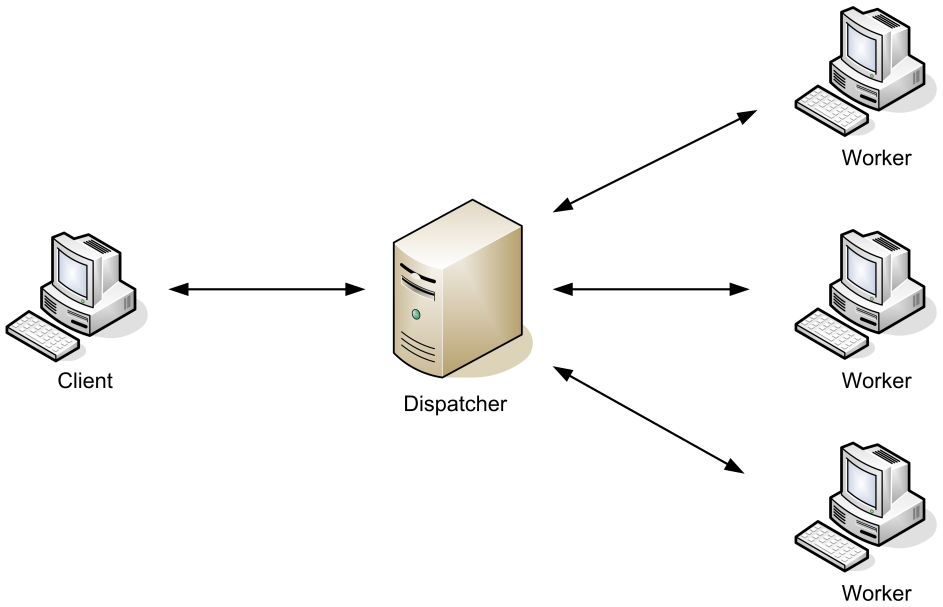

Figure 7.6: *L'architecture du système PovRay*

1. Le grid computing permet de concentrer les puissances de calcul d'ordinateurs répartis dans le monde et reliés ensemble via le réseau internet afin de créer un super-calculateur.

D'un point de vue abstrait, l'architecture comporte trois types de composants :

- **Le client :** c'est le point d"entrée du système. Il permet de soumettre des tâches au grid, de les monitorer, et de récupérer les résultats de leur exécution.
- **L'exécuteur (Worker) :** c'est une machine volontaire qui se propose de faire du rendu d'images via PovRay.
- **Le serveur (Dispatcher) :** c'est une unité de coordination. Le serveur gère les requêtes des clients pour les distribuer vers les machines workers.

7.4.2 Implémentation des composants

Ce qui est intéressant avec cette architecture, c'est qu'il n'est pas évident de choisir la meilleure et que cela motive évidemment l'approche du framework proposé.

Dans le cadre de notre framework, nous devons définir des stratégies de communications différentes entre les composants qui seraient succeptibles de rendre le système plus efficace, comme nous allons le voir ci-après.

Dans le cas présent il faut veiller à avoir des stratégies à la louche, le but n'est pas d'implémenter le TP, il faut émuler pour donner une impression.

Au niveau des différentes stratégies on peut aisément en identifier quelques unes :

1. Au niveau de la fragmentation de l'image :
   (a) Est-ce le dispatcher qui fragmente l'image reçue par le client
   (b) Est-ce le client qui fragmente puis envoie ses morceaux au dispatcher
2. Au niveau de retour du résultat :
   (a) Via le dispatcher
   (b) Directement au client

La figure 7.7 montre les composants de la stratégie 1a et 2a.

Pour représenter cette stratégie nous pouvons également utiliser un diagramme de séquence dont le contenu se trouve à la figure 7.8.

La figure 7.9 quant à elle montre les composants de la stratégie 1a et 2b, dans ce cas c'est le worker qui retourne le résultat au client.

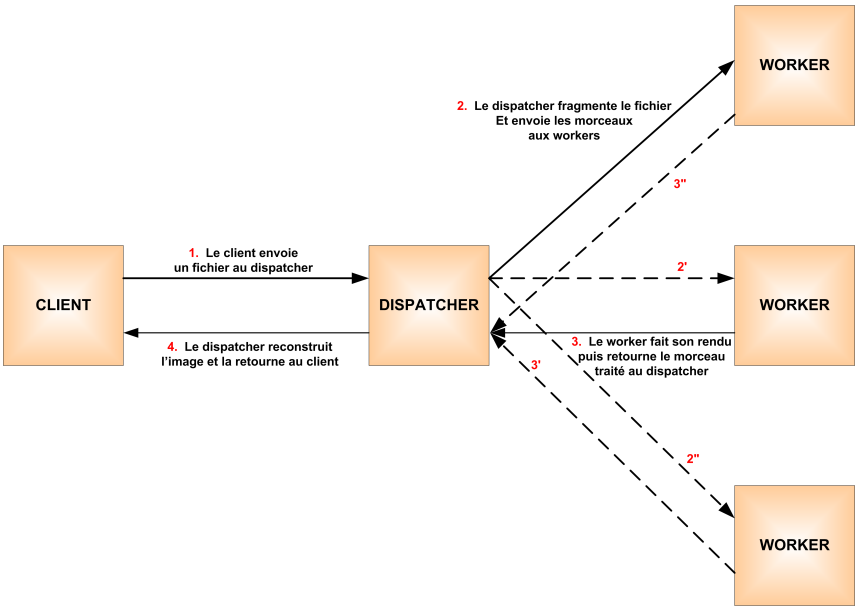

Figure 7.7: *Les composants représentant la stratégie 1a et 2a*

Le scénario de la stratégie 1a et 2a

```
package scenario;

import java.rmi.RemoteException;

import factory.Machine;
import co.Choreography;
import collections.Components;
import component.Parameters;
import static tools.Tools.*;

public class PovRay_S1 extends Choreography {


        public PovRay_S1() {

                Machine M1 = createMachine("M1", "127.0.0.1");

                Parameters pWorker = new Parameters();
                // Creation de 3 composants de type "Worker" sur la machine M1
                Components W = createComponents(M1, 3, "Test.Worker", pWorker)
                    ;

                Parameters pDisp = new Parameters();
                pDisp.add("w1", W.get(1));
                pDisp.add("w2", W.get(2));
                pDisp.add("w3", W.get(3));
```

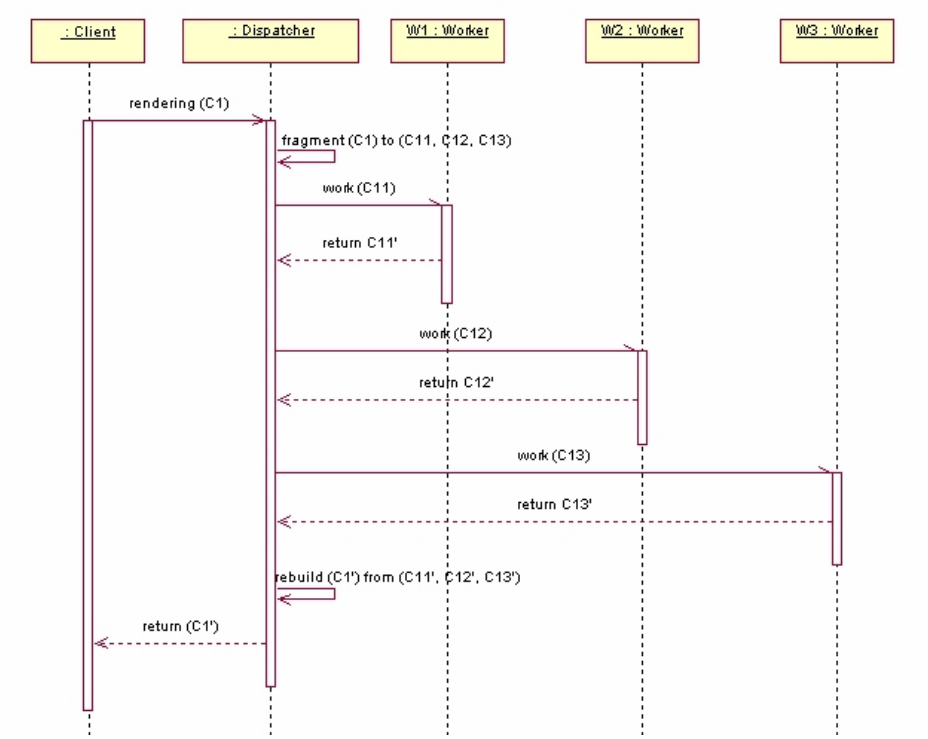

Figure 7.8: *Diagramme de séquence de la stratégie 1a et 2a*

```
// Creation de 1 composant de type "Client" sur la machine M2
Components D = createComponents(M1, 1, "Test.Dispatcher",
    pDisp);


// Creation de 1 composant de type "Client" sur la machine M1
Components C = createComponents(M1, 1, "Test.ClientS2", null);


connect(C.get(1),D.get(1)); // connect C to D


W.startOneway();
D.startOneway();
C.startOneway();


pause(30000); // Wait X ms.
        // Stop the system
        try {
                M1.stop();
        } catch (RemoteException e) {
                e.printStackTrace();
        }
```

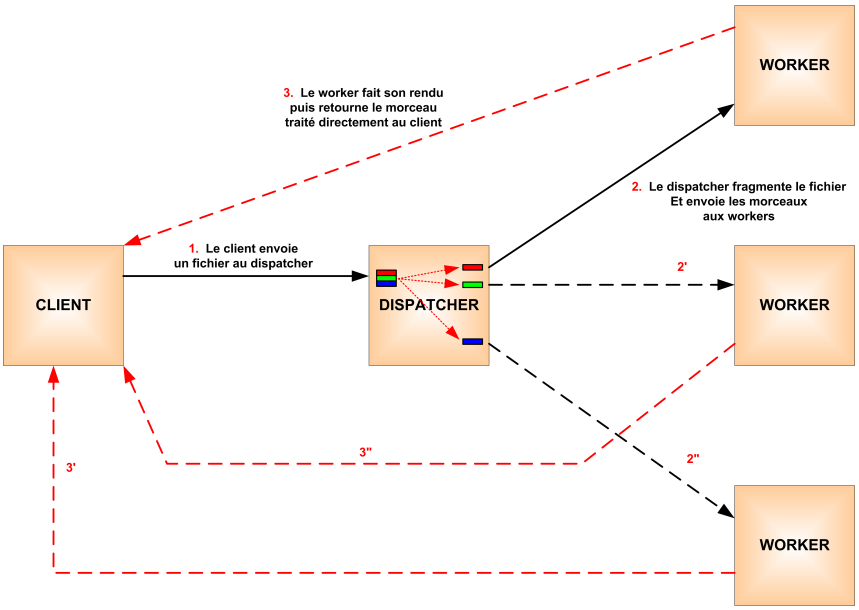

Figure 7.9: *Les composants représentant la stratégie 1a et 2b*

```
        }

}
```

Le Client avec stratégie 1a et 2a

```
package Case3_PovRAY;

import java.rmi.RemoteException;
import measure.*;
import static tools.Tools.*;
import component.*;

public class ClientS1Impl extends GenericComponent implements Client {

        public ClientS1Impl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);


                chrono = new Chrono(this,"MyChrono");
                Memory mem = new Memory(this,"MyMemory");
                while (RUNNING) {

                        System.out.println("Size of Image before rendering =
                            3000");
```

```
                        mem.start();
                        chrono.start();
                        image = nextComponent.requestS(createString(3000));
                        chrono.stop();
                        chrono.save();
                        chrono.display();
                        mem.stop();
                        mem.save();
                        System.out.println("Size of Image after rendering = "
                            + image.length());


                        // Time Loop = 1/2 sec.
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }

                }

        }

        public void request(String s) throws RemoteException {
                super.request(s);
                super.cptRequest.display();
        }

}
```

Le scénario de la stratégie 1a et 2b

```
package scenario;

import java.rmi.RemoteException;

import factory.Machine;
import co.Choreography;
import collections.Components;
import component.Parameters;
import static tools.Tools.*;

public class PovRay_S2 extends Choreography {


        public PovRay_S2() {

                Machine M1 = createMachine("M1", "127.0.0.1");

                Parameters pWorker = new Parameters();
                // Creation de 3 composants de type "Worker" sur la machine M1
                Components W = createComponents(M1, 3, "Test.Worker", pWorker)
                    ;

                Parameters pDisp = new Parameters();
                pDisp.add("w1", W.get(1));
                pDisp.add("w2", W.get(2));
```

```
                pDisp.add("w3", W.get(3));

                // Creation de 1 composant de type "Client" sur la machine M2
                Components D = createComponents(M1, 1, "Test.Dispatcher",
                    pDisp);


                // Creation de 1 composant de type "Client" sur la machine M1
                Components C = createComponents(M1, 1, "Test.ClientS2", null);


                connect(C.get(1),D.get(1)); // connect C to D
                W.connect(C.get(1)); // connect all W to C

                W.startOneway();
                D.startOneway();
                C.startOneway();


                pause(30000); // Wait X ms.
                        // Stop the system
                        try {
                                M1.stop();
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }


        }

}
```

Le Client avec stratégie 1a et 2b

```
package Case3_PovRAY;

import java.rmi.RemoteException;
import measure.*;
import component.*;

import static tools.Tools.*;

public class ClientS2Impl extends GenericComponent implements Client {

        public String image = "";
        public Chrono chrono;

        public ClientS2Impl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);

                chrono = new Chrono(this,"MyChrono");
                Memory mem = new Memory(this,"MyMemory");
                while (RUNNING) {
                        mem.start();
```

```
                        chrono.start();

                        System.out.println("Size of Image before rendering =
                            3000");
                        nextComponent.request(createString(3000));

                        mem.stop();
                        mem.save();

                        pause(500);

                }

        }

        public synchronized void request(String s) throws RemoteException {
                // Reconstitution de l'image
                image = image + s;
                if (image.length() >= 3300) {
                        System.out.println("Size of Image after rendering ="
                            + image.length());
                        chrono.stop();
                        chrono.save();
                        chrono.display();
                        image = "";
                }

        }

}
```

Le Dispatcher

```
package Case3_PovRAY;

import java.rmi.RemoteException;
import measure.*;
import component.*;
import static tools.Tools.*;

import static tools.Tools.*;

public class DispatcherImpl extends GenericComponent implements Dispatcher {

        public Component w1, w2, w3;

        public Chrono chrono;

        public DispatcherImpl() throws RemoteException {

        }

        public void start(Params p) throws RemoteException {
                super.start(p);
                w1 = (Component) parameters.resolve("w1");
                w2 = (Component) parameters.resolve("w2");
                w3 = (Component) parameters.resolve("w3");
                chrono = new Chrono(this, "MyMeasure");
```

```
        }

        public void request(String s) throws RemoteException {
                // On divise l'image en 3
                String s1 = createString(s.length() / 3);
                String s2 = createString(s.length() / 3);
                String s3 = createString(s.length() / 3);
                chrono.start();
                w1.request(s1);
                w2.request(s2);
                w3.request(s3);
                chrono.stop();
                chrono.save();
        }

        public String requestS(String s) throws RemoteException {
                // On divise l'image en 3
                String s1 = createString(s.length() / 3);
                String s2 = createString(s.length() / 3);
                String s3 = createString(s.length() / 3);
                chrono.start();
                String r1 = w1.requestS(s1);
                String r2 = w2.requestS(s2);
                String r3 = w3.requestS(s3);
                chrono.stop();
                chrono.save();

                return r1 + r2 + r3;
        }

}
```

Le Worker

```
package Case3_PovRAY;

import java.rmi.RemoteException;

import component.*;
import static tools.Tools.*;


public class WorkerImpl extends GenericComponent implements Worker {

        public WorkerImpl() throws RemoteException {

        }

        public void request(String s) throws RemoteException {
                // On traite l'image reçue
                cpuFloat(100);
                String s1 = s + createString(s.length()/10);

                nextComponent.request(s1);
        }

        public String requestS(String s) throws RemoteException {
                // On traite l'image reçue
```

```
                cpuFloat(100);

                String s1 = s + createString(s.length()/10);

                return s1;
        }
}
```

7.4.3 Résultats des tests

Les résultats des mesures du système se trouvent à la figure 7.10, et la figure 7.11 compare les chronos du client par rapport à ceux du dispatcher.

	A	B	C	D	E	F
1	M1	Test.Client(1)	MyMeasure	Chrono	ms	156
2	M1	Test.Client(1)	MyMeasure	Chrono	ms	94
3	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
4	M1	Test.Client(1)	MyMeasure	Chrono	ms	94
5	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
6	M1	Test.Client(1)	MyMeasure	Chrono	ms	94
7	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
8	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
9	M1	Test.Client(1)	MyMeasure	Chrono	ms	109
10	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
30	M1	Test.Client(1)	MyMeasure	Chrono	ms	125
31	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
32	M1	Test.Client(1)	MyMeasure	Chrono	ms	78
33	M1	Test.Client(1)	MyMeasure	Chrono	ms	78

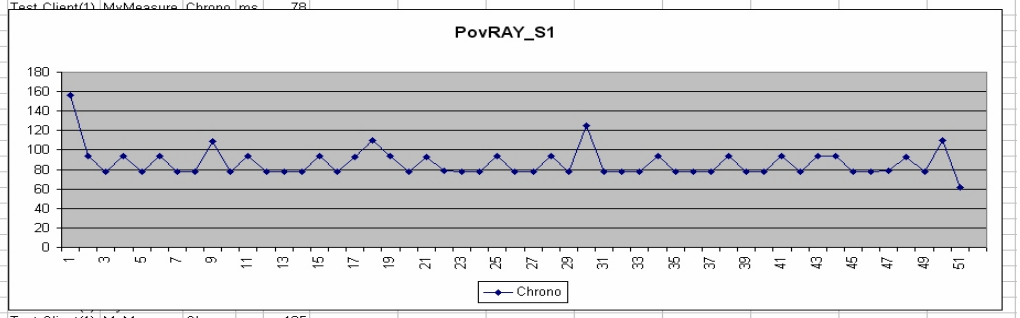

Figure 7.10: *Résultats de la stratégie 1a et 2a*

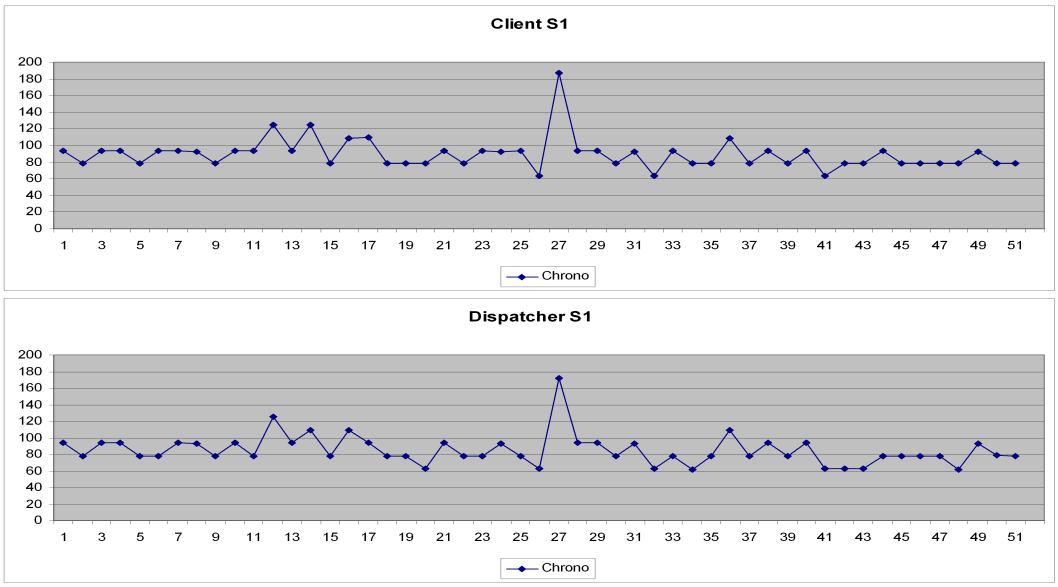

Figure 7.11: *Comparatif Client - Dispatcher avec stratégie 1a et 2a*

CHAPITRE 8

Conclusion et perspective

8.1 Conclusion

La création d'un framework, que ce soit en Java ou dans une autre langage, est toujours difficile parce qu'il y a toujours des concepts qui manquent à un moment ou un autre de son utilisation. Le framework proposé est certe très limité, car comme nous le rappelons nous avons fait souvent les choix les plus simples [1], mais ce que nous avons fait est un "'proof of concept"' c'est-à-dire vérifier la faisabilité et l'application de concepts.

Le but principal de ce mémoire était de proposer des outils qui permettent de décrire et d'émuler, de manière le plus simple possible, un petit système distribué et de pouvoir faire quelques mesures de performances. Ceci afin de pouvoir les exploiter et d'en tirer des conclusions en se basant sur nos connaissances du moment.

Bien que le système proposé soit très simple, il est tout de même à souligner que les concepts mis en place pour émuler un système distribué et pouvoir faire quelques mesures de performances foncionne avec le framework, et ce même si celui-ci est limité. Les fichiers résultats demandent quelques manipulations manuelles afin d'exploiter les mesures, mais cela fait partie des hypothèses que nous nous sommes posés en début de travail.

Le framework est écrit en Java, ce qui a permis de l'implémenter en orienté objet

1. ceci est dû au contexte très particulier dans lequel s'est fait ce travail de fin d'études

et a permis d'utiliser facilement certains patterns comme par exemple la fabrique d'objets distants. Le mémoire n'était pas basé sur des études de cas mais ceux-ci devaient éliciter les concepts à mettre en place pour implémenter le framework.

Les tests effectués avec le framework ont mis en évidence le fait que les concepts mis en place suffisent à simuler un bon nombre de prototypes de SD dont notamment les trois études de cas.

Le choix de créer un framework qui mélange du concret et de l'abstrait et sans doute une bonne chose en ce qui concerne la rapidité de développement des composants, mais cela reste tout de même assez lourd et cela oblige aussi à connaître la syntaxe du langage Java.

8.2 Perspectives

La version du framework n'est pas labelisé (pas de version) et donc celui-ci est vraiment une version bêta. Cela reste donc bien dans le cadre d'un développement d'un outil pédagogique et sans doute que pas mal d'améliorations du système proposé peuvent rendre le système plus interressant à savoir :

- La framework est écrit en Java/RMI, il va de soit que Java/RMI n'est pas le plus performant des middlewares et il serait intéressant d'implémenter le framework avec un autre langage, et sans doute aussi avec un autre middleware comme CORBA afin de faire des comparaisons non plus entre des scénarios différents mais bien entre différents middlewares. On pourrait dès lors faire en sorte que le framework soit indépendant du middleware, ce qui augmenterait sa généricité
- Le CO se lance par ligne de commande et il aurait été plus convivial d'avoir une IHM (*Interface Home Machine*) qui permettrait non seulement de choisir des scénarios dans une liste mais aussi de pouvoir rapatrier les fichiers résultats directement via le CO. On pourrait également envisager que les résultats soient directement afficher dans l'IHM sous forme de courbes
- Dans le chapitre des hypothèses de travail, nous expliquons qu'une des démarches de travail possible était de faire un interpréteur de fichier (*avec les outils LEX et YACC*), bien que cette piste ait été abandonnée assez vite en raison de sa plus grande complexité, il serait très interessant d'en faire l'étude et la conception
- Une autre voie interessante aurait été de définir de manière formel un langage abstrait avec toute sa syntaxe, sa grammaire ainsi que toutes ses fonctionnalités (instructions, fonctions mathématiques, etc.) qui serviraient à la simulation de SD mais cette fois en une vraie simulation et non plus en mélangeant du concret et de l'abstrait, ce langage permettrait aussi de faire des mesures de

performances

- Le format des fichiers résultats est CSV (*Comma Separated Values*), il va de soit que cela ne devrait pas être forcément le cas, CSV à été choisi pour insérer les valeurs plus facilement dans un tableur comme Excel, mais il serait interessant d'utiliser des fichiers XML, ceux-ci pourraient alors être utilisé avec des feuilles de styles et permettre de générer des graphes facilement et de manière automatique
- Le déploiement des différents composants du système, c'est-à-dire la copie des fichiers (code, souche, etc.) sur les différentes machines du système est manuelle, une procédure de déploiement automatique serait la bienvenue
- Enfin, pour le système de mesures nous avons utilisé un simple héritage de classe et nous savons que l'héritage doit-être éviter dans des systèmes qui se veulent extensibles, il est préférable d'utiliser la composition et donc il serait intéressant de changer cet héritage de classe par une composition (voir ***[FF05]*** pour une excellente explication sur la composition par rapport à l'héritage)

Toutes ces perspectives n'ont pas été étudiées dans le cadre de ce mémoire, et à la question : *"'Feriez-vous la même chose si vous deviez tout recommencer ?"'*, nous pouvons répondre par la négative car il est certain que les voies les plus simples ne seraient pas forcément prises, mais je ne doute pas que d'autres étudiants prennent ces voies pour apporter leurs pierres à l'édifice.

Bibliographie

[Ber07] Bersini H., *L'orienté objet*, Eyrolles, 3ème ed., France, 2007, ISBN 978-2-212-12084-8

[CDK03] Coulouris G., Dollimore J., Kindberg T., *Distributed Systems, Concepts and Desing*, Addison-Wesley, third ed., 6th impression 2003, ISBN 0-201-61918-0

[Dar02] Ian F. Darwin , *Java en action*, O'Reilly, Paris, France, 2002, ISBN 2-84177-203-9

[Dou07] Doudoux J.M., *Développons en Java avec Eclipse*, http ://jmdoudoux.developpez.com/cours/developpons/eclipse/, (version 0.80.1 decembre 2008)(Dernier accès 15/01/09)

[Eng04] Engelbert V., *Conceptions des systèmes distribués et coopératifs*, FUNDP, 2004

[FF05] Freeman E., Freeman E., *Head First Design Patterns*, O'Reilly, USA, 2005, ISBN 0-596-00712-4

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.,*Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN 0-201-63361-2

[GCL05] rundy J.C., Cai Y., Liu A., *SoftArch/MTE : Generating Distributed System Test-beds from High-level Software Architecture Descriptions*, Automated Software Engineering, vol. 12, no. 1, January 2005, pp. 5-39, Kluwer Academic Publishers

[Gom00] Gomaa H., *Designing Concurrent, Distributed, and Real-Time applications with UML*, Addison-Wesley, third ed., impression 2003, ISBN 0-201-65793-7

[Kolp07] Kolp M., *UML en pratique : modélisation avec Rational Rose*, TechnoFutur TIC, UCL-Isis

CHAPITRE 9

Annexe A - Description de classes du Framework

9.1 Le chef d'orchestre et ses classes

9.1.1 La classe CO

```
package co;

import java.io.IOException;
import java.rmi.*;

import static co.ScenarioPlayer.*;



public class CO {

    /**
     * This method is the main method
     *
     */
    public static void main(String[] argv) {

        String scenario = ""; // Name of the scenario

        if (System.getSecurityManager() == null) {
                        System.out.println("CO : Start the Security Manager
                            ...");
                        System.setSecurityManager(new RMISecurityManager());
        }
                else {
                        System.out.println("CO : The Security Manager is
                            already started ...");
                }
```

```
        if (argv.length != 1) {
                System.out.println("CO : No scenario in argument ...");

        }
        else {

                scenario = argv[0];
                System.out.println("CO : Starting the scenario " + scenario);
                playScenario (scenario);
        }


    }
}
```

9.1.2 La classe ScenarioPlayer

```
package co;

import component.Component;



public class ScenarioPlayer {

        private ScenarioPlayer() {};

        public static void playScenario (String scenario) {

        Object obj = null;


        if (scenario != null) {
            System.out.println("ScenarioPlayer(" + scenario + ")");
        } else {
            System.out.println("Scenario name is Null");
        }

        try {
             Class myClass = Class.forName(scenario);
            // New instance of "scenario"
            try {
                obj = myClass.newInstance();
            } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...");
            } catch (InstantiationException ex) {
                ex.printStackTrace();
                System.out.println("InstantiationException ...");
            }
        } catch (ClassNotFoundException e) {
            System.out.println("Scenario " + scenario + " not Found !");
        }


        }
```

```
}
```

9.1.3 La classe Choreography

```
package co;

import java.rmi.Naming;
import java.rmi.RemoteException;

import component.Component;
import component.Parameters;
import collections.Components;
import collections.Machines;

import factory.Machine;

/**
 * @author JC
 *
 */
public class Choreography {

        /**
         * Cette méthode permet d'obtenir la référence de l'objet "Machine"
             sur
         * l'ordinateur cible
         *
         * @param ip
         *            adresse de la machine cible
         * @param port
         *            numéro de port de la machine cible
         * @return Machine
         *
         */
        public static Machine createMachine(String ip, int port) {
                Machine m = null;
                String hostName = "";

                try {
                        // bind Machine object to Machine object
                        m = (Machine) Naming.lookup("//" + ip + ":" + port + "
                            /Machine");
                        hostName = m.getHostName();
                } catch (Exception e) {
                        System.out.println("createMachine Exception : " + e);
                        System.exit(0);
                }
                if (hostName == null)
                        hostName = ip;
                // Add this machine to the Machines collection
                Machines.addMachine(hostName, m);
                // Assigne un nom de machine
                try {
                        m.setMachineName(hostName);
                } catch (RemoteException e) {

                        System.out.println("setMachineName Exception : " + e);
                }
```

```
                return m;

        }

        public static Machine createMachine(String ip) {
                Machine m = null;
                String hostName = "";

                try {
                        // bind Machine object to Machine object
                        m = (Machine) Naming.lookup("//" + ip + ":" + "1099" +
                             "/Machine");
                        hostName = m.getHostName();
                } catch (Exception e) {
                        System.out.println("createMachine Exception : " + e);
                        System.exit(0);
                }
                if (hostName == null)
                        hostName = ip;
                // Add this machine to the Machines collection
                Machines.addMachine(hostName, m);
                // Assigne un nom de machine
                try {
                        m.setMachineName(hostName);
                } catch (RemoteException e) {
                        System.out.println("setMachineName Exception : " + e);
                }
                return m;

        }

        /**
         * Cette méthode permet d'obtenir la référence de l'objet "Machine"
             sur
         * l'ordinateur cible
         *
         * @param name
         *            nom de la machine cible
         * @param ip
         *            adresse de la machine cible
         * @param port
         *            numéro de port de la machine cible
         * @return Machine
         *
         */
        public static Machine createMachine(String name, String ip, int port)
            {

                Machine m = null;

                try {
                        // bind Machine object to Machine object
                        m = (Machine) Naming.lookup("//" + ip + ":" + port + "
                            /Machine");
                } catch (Exception e) {
                        System.out.println("createMachine Exception : " + e);
                        System.exit(0);
                }
```

```
                if (name == null)
                        name = ip;
                // Ajout de cette machine dans la collection de Machines
                Machines.addMachine(name, m);
                // Assigne un nom de machine
                try {
                        m.setMachineName(name);
                } catch (RemoteException e) {

                        System.out.println("setMachineName Exception : " + e);
                }

                return m;

        }

        public static Machine createMachine(String name, String ip) {

                Machine m = null;

                try {
                        // bind Machine object to Machine object
                        m = (Machine) Naming.lookup("//" + ip + ":" + "1099" +
                             "/Machine");
                } catch (Exception e) {
                        System.out.println("createMachine Exception : " + e);
                        System.exit(0);
                }
                if (name == null)
                        name = ip;
                // Ajout de cette machine dans la collection de Machines
                Machines.addMachine(name, m);
                // Assigne un nom de machine
                try {
                        m.setMachineName(name);
                } catch (RemoteException e) {

                        System.out.println("setMachineName Exception : " + e);
                }
                return m;

        }

        public static Components createComponents(Machine machine, int number,
                        String componentName, Parameters params) {

                Components comp = new Components();
                comp.setMachine(machine);
                comp.setParameters(params);
                Component component = null;

                for (int i = 0; i < number; i++) {
                        try {
                                component = (Component) machine.
                                    createComponent(i + 1,
                                                    componentName);
                        } catch (Exception e) {
```

```
                                System.out.println("createComponents Exception
                                    : " + e);
                        }
                        // add the component into the collection
                        comp.collection.add(i, component);
                }

                return comp;
        }

        public void connect(Component firstComponent, Component
            secondComponent) {

                try {
                        // invoke the method connect() on the remote object "
                            firstComponent"
                        firstComponent.connect(secondComponent);
                } catch (RemoteException e) {
                        e.printStackTrace();
                }
        }

}
```

9.1.4 La classe StartOneway

```
package co;

import java.rmi.RemoteException;

import component.Component;
import component.Parameters;

public class StartOneway extends Thread {

        private Component comp;

        private Parameters params;

        public StartOneway(Component component, Parameters params) {
                this.comp = component;
                this.params = params;
                this.start();
        }

        public void run() {
                try {
                        // invoke the method start() on the remote object "get
                            (i)"
                        comp.start(params);
                } catch (RemoteException e) {
                        e.printStackTrace();
                }
        }
}
```

9.2 La fabrique de composants

9.2.1 La classe RMILauncher

```
package factory;

import java.rmi.*;

public class RMILauncher {

        public static void main(String[] argv) {

                String serverName = "";
                String serverIP = "";

                try {
                        System.out.print("Start the RMI Registry on port 1099 ...");
                        if (java.rmi.registry.LocateRegistry.getRegistry() == null) {
                                java.rmi.registry.LocateRegistry.createRegistry(1099);
                                System.out.println("OK");
                        } else {
                                System.out.println("WARNING");
                                System.out.println("The RMI Registry is already started ...");
                        }
                } catch (RemoteException e1) {
                        System.out.println("RMI Registry Error: " + e1.getMessage());
                }

                try {
                        System.out.print("Start the Security Manager ...");
                        if (System.getSecurityManager() == null) {
                                System.setSecurityManager(new java.rmi.RMISecurityManager());
                                System.out.println("OK");
                        } else {
                                System.out.println("WARNING");
                                System.out
                                                .println("The Security Manager is already started ...");
                        }

                } catch (Exception e) {
                        System.out.println("Security Manager Error: " + e.getMessage());
                }

                try {
                        // IP Address and HostName
                        serverName = java.net.InetAddress.getLocalHost().getHostName();
                        serverIP = java.net.InetAddress.getLocalHost().getHostAddress();
```

```
                        MachineImpl fabrique = new MachineImpl("Machine");

                        System.out.println("Machine on " + serverName + "(" +
                            serverIP
                                        + ") is ready");
                } catch (Exception e) {
                        System.out.println("Machine Error on " + serverName +
                            "("
                                        + serverIP + "): " + e.getMessage());
                }
        }
}
```

9.2.2 La classe MachineImpl et son interface

```
package factory;

import component.Component;

public interface Machine extends java.rmi.Remote {

        public void setMachineName(String machineName)
                        throws java.rmi.RemoteException;

        public String getHostName() throws java.rmi.RemoteException;

        public Component createComponent(String name)
                        throws java.rmi.RemoteException;

        public Component createComponent(int cID, String name)
                        throws java.rmi.RemoteException;

        public void stop() throws java.rmi.RemoteException;

}
```

```
package factory;

import component.Component;
import java.lang.reflect.Field;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import collections.MachineComponents;


public class MachineImpl extends UnicastRemoteObject implements Machine {


        public String machineName = "";
        public String machineNameToRegistry = "";
        public MachineComponents collection = new MachineComponents();

        public MachineImpl(String name) throws RemoteException {
                super();
```

```
                this.machineNameToRegistry = name;

                try {
                        // name : nom spécifié dans l'annuaire
                        Naming.rebind(name, this);

                } catch (Exception e) {
                        if (e instanceof RemoteException) {
                                throw (RemoteException) e;
                        } else {
                                throw new RemoteException(e.getMessage());
                        }
                }
                // clear the collection
                collection.clear();
        }

        /**
         * @param machineName  the machineName to set
         * @uml.property  name="machineName"
         */
        public void setMachineName(String machineName) throws RemoteException
            {
                this.machineName = machineName;
        }

        public String getHostName() throws RemoteException {
                String serverName = "";
                try {
                        serverName = java.net.InetAddress.getLocalHost().
                            getHostName();
                } catch (UnknownHostException e) {
                        e.printStackTrace();
                }
                return serverName;
        }

        public Component createComponent(String name) throws RemoteException {

                String className = name + "Impl";
                String cName = name + "(1)";
                Object obj = null;
                Component c = null;


                if (className != null) {
                        System.out.println(machineName + ".createComponent(" +
                             className
                                        + ")==>" + cName );
                } else {
                        System.out.println("Component name is Null on " +
                            machineName);
                }

                try {
                        Class myClass = Class.forName(className);
                        // New instance of "className"
                        try {
```

```
                obj = myClass.newInstance();
        } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
        } catch (InstantiationException ex) {
                ex.printStackTrace();
                System.out.println("InstantiationException ...
                    ");
        }

        Field componentID = null;
        try {
                // return the reference of the object attribut
                    "cID"
                componentID = obj.getClass().getField("
                    componentID");
        } catch (SecurityException ex) {
                ex.printStackTrace();
                System.out.println("FieldSecurityException ...
                    ");
        } catch (NoSuchFieldException ex) {
                ex.printStackTrace();
                System.out.println("NoSuchFieldException ...");
                    ;
        }
        try {
                // set the cID
                componentID.setInt(obj, 1);
        } catch (IllegalArgumentException ex) {
                ex.printStackTrace();
                System.out.println("IllegalArgumentException
                    ...");
        } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
        }

        Field componentName = null;
        try {
                // return the reference of the object attribut
                    "cID"
                componentName = obj.getClass().getField("
                    componentName");
        } catch (SecurityException ex) {
                ex.printStackTrace();
                System.out.println("FieldSecurityException ...
                    ");
        } catch (NoSuchFieldException ex) {
                ex.printStackTrace();
                System.out.println("NoSuchFieldException ...");
                    ;
        }
        try {
                // set the componentName
                componentName.set(obj, cName);
        } catch (IllegalArgumentException ex) {
```

```
                                ex.printStackTrace();
                                System.out.println("IllegalArgumentException ...");
                        } catch (IllegalAccessException ex) {
                                ex.printStackTrace();
                                System.out.println("IllegalAccessException ...");
                        }

                        Field machineID = null;
                        try {
                                // return the reference of the object attribut "machineID"
                                machineID = obj.getClass().getField("machineID");
                        } catch (SecurityException ex) {
                                ex.printStackTrace();
                                System.out.println("FieldSecurityException ...");
                        } catch (NoSuchFieldException ex) {
                                ex.printStackTrace();
                                System.out.println("NoSuchFieldException ...");
                        }
                        try {
                                // set the machineID
                                machineID.set(obj, machineName);
                        } catch (IllegalArgumentException ex) {
                                ex.printStackTrace();
                                System.out.println("IllegalArgumentException ...");
                        } catch (IllegalAccessException ex) {
                                ex.printStackTrace();
                                System.out.println("IllegalAccessException ...");
                        }

                } catch (ClassNotFoundException e) {
                        System.out.println("Component " + className + " not Found on "
                                        + machineName + "!");
                }
                c = (Component) obj;

                // ajout du composant dans la collection
                collection.addComponent(c);


                return c;

        }

        public Component createComponent(int cID, String name)
                        throws RemoteException {
                String className = name + "Impl";
                String cName = name + "(" + cID + ")";
                Object obj = null;
                Component c = null;
```

```
if (className != null) {
        System.out.println(machineName + ".createComponent(" +
             className
                        + ")==>" + cName );
} else {
        System.out.println("Component name is Null on " +
            machineName);
}

try {
        Class myClass = Class.forName(className);
        // New instance of "className"
        try {
                obj = myClass.newInstance();
        } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
        } catch (InstantiationException ex) {
                ex.printStackTrace();
                System.out.println("InstantiationException ...
                    ");
        }

        Field componentID = null;
        try {
                // return the reference of the object attribut
                     "cID"
                componentID = obj.getClass().getField("
                    componentID");
        } catch (SecurityException ex) {
                ex.printStackTrace();
                System.out.println("FieldSecurityException ...
                    ");
        } catch (NoSuchFieldException ex) {
                ex.printStackTrace();
                System.out.println("NoSuchFieldException ...")
                    ;
        }
        try {
                // set the cID
                componentID.setInt(obj, cID);
        } catch (IllegalArgumentException ex) {
                ex.printStackTrace();
                System.out.println("IllegalArgumentException
                    ...");
        } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
        }

        Field componentName = null;
        try {
                // return the reference of the object attribut
                     "cID"
```

```
                componentName = obj.getClass().getField("
                    componentName");
            } catch (SecurityException ex) {
                ex.printStackTrace();
                System.out.println("FieldSecurityException ...
                    ");
            } catch (NoSuchFieldException ex) {
                ex.printStackTrace();
                System.out.println("NoSuchFieldException ...")
                    ;
            }
            try {
                // set the componentName
                componentName.set(obj, cName);
            } catch (IllegalArgumentException ex) {
                ex.printStackTrace();
                System.out.println("IllegalArgumentException
                    ...");
            } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
            }


            Field machineID = null;
            try {
                // return the reference of the object attribut
                     "machineID"
                machineID = obj.getClass().getField("machineID
                    ");
            } catch (SecurityException ex) {
                ex.printStackTrace();
                System.out.println("FieldSecurityException ...
                    ");
            } catch (NoSuchFieldException ex) {
                ex.printStackTrace();
                System.out.println("NoSuchFieldException ...")
                    ;
            }
            try {
                // set the machineID
                machineID.set(obj, machineName);
            } catch (IllegalArgumentException ex) {
                ex.printStackTrace();
                System.out.println("IllegalArgumentException
                    ...");
            } catch (IllegalAccessException ex) {
                ex.printStackTrace();
                System.out.println("IllegalAccessException ...
                    ");
            }

        } catch (ClassNotFoundException e) {
            System.out.println("Component " + className + " not
                Found on "
                    + machineName + "!");
        }
```

```
                c = (Component) obj;

                // ajout du composant dans la collection
                collection.addComponent(c);

                return c;
        }

        public void stop() throws RemoteException {


                // On fait appel à la méthode stop() de tous les éléments de
                    la collection de composants
                for (int i = 0; i < collection.size(); i++) {

                        Component cToCall = (Component) collection.
                            getComponent(i);
                        try {
                                // invoke the method stop() on the remote
                                    object "get(i)"
                                cToCall.stop();
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }

        }
}
```

9.3 La librairie de simulation

9.3.1 La classe Tools

```
package tools;


import java.util.Random;

public class Tools {

        private Tools() {
                //      No Instanciation
        }

        /**
         * Cette méthode permet d'obtenir une probabilité moyenne Ex. : Si i =
              66,
         * alors on retournera True dans 2/3 des cas, 66% (en moyenne) Si i =
             50,
         * alors on retournera True dans 1 cas sur 2, 50% (en moyenne)
         *
         * @param i
         *            valeur comprise entre 0 et 100
         * @return boolean
         */
        public static boolean proba(int i) {
```

```
                if (i > 0 && i < 100) {
                        // Créer un nombre aléatoire entre 0 et 1
                        Random r1 = new Random();
                        int r2 = (int) (r1.nextDouble() * 100);
                        // Retourner True si value <= i
                        return (r2 <= i);
                } else
                        return false; // Parce que i n'est pas compris entre 0
                             et 100
        }


        /**
         * Cette méthode permet de simuler une charge cpu, en faisant une
             boucle qui
         * calcule une fonction cosinus
         *
         * @param charge
         *            Longueur de la boucle, charge à simuler
         * @return
         */
        public static void cpuInt(int charge) {
                int result = 0;
                for (int i = 1; i <= charge * 100; i++) {
                        result = (int) Math.abs(Math.cos(i));
                }
        }

        /**
         * Cette méthode permet de simuler une charge cpu, en faisant une
             boucle qui
         * calcule une fonction cosinus
         *
         * @param charge
         *            Longueur de la boucle, charge à simuler
         * @return
         */
        public static void cpuFloat(int charge) {
                double result, res1, res2, res3, res4 = 0.0;
                for (int i = 1; i <= charge * 100; i++) {
                        //      Créer un nombre aléatoire entre 0 et 1
                        Random r1 = new Random();
                        int r2 = (int) (r1.nextDouble() );
                        res1 = Math.cos(r2);
                        res2 = Math.sin(res1);
                        res3 = Math.cos(res2);
                        res4 = Math.sin(res3);
                        result = Math.sqrt(res4);
                }
        }

        /**
         * Cette méthode permet de simuler une charge cpu, en créant une
             chaîne de 100 * charge
         * caractères puis en transformant celle-ci en MAJUSCULE
         *
         * @param charge
         *            Longueur de la chaîne, charge à simuler
```

```
         * @return
         */
        public static void cpuString(int charge) {

                String s = createString(charge * 100);
                s.toUpperCase();

        }
        /**
         * Cette méthode permet de construire une chaîne de caractères
         *
         * @param nbrChar
         *            Longueur de la chaîne à construire
         * @return String
         */
        public static String createString(int nbrChar) {
                StringBuffer sb = new StringBuffer(nbrChar);
                for (int i = 0; i < nbrChar; i++) {
                        sb.append("@");
                }
                return sb.toString();
        }

}
```

9.3.2 La classe Simul

```
package tools;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Simul {

        // File name for the writing method
        public String theFileW = "";

        // The constructor
        public Simul() {
                this.createTempoFile();
        }



        /**
         * Cette méthode permet de faire un accès disque en lecture et/ou
             écriture.
         * Elle permet également de simuler l'accès à une Base de Données
         *
         * @param value
         *            Nombre de lecture et/ou écriture
         * @param mode
         *            Mode d'accès 'r' pour la lecture et 'w' pour l'écriture
         * @return
```

```
	 */
	public void disk(int value, char mode) {

		switch (mode) {
		case 'r':
		case 'R':

			String leFichier = System.getProperty("user.dir") + "
			    \\tmpReadFile\\"
					+ "Read_TmpFile.txt";
			System.out.println("Reading(" + value + ") of the file
			    "
					+ leFichier);
			try {
				FileReader fichier = new FileReader(leFichier)
				    ;
				// Effectuer la lecture d'octet en fonction de
				     @value
				for (int i = 1; i <= value; i++) {
					fichier.read();
				}
				fichier.close();
			} catch (FileNotFoundException e) {
				// TODO Bloc catch auto-généré
				e.printStackTrace();
			} catch (IOException e) {
				// TODO Bloc catch auto-généré
				e.printStackTrace();
			}

			break;
		case 'w':
		case 'W':
			// Open and writing in the file 'theFileW'
			System.out.println(theFileW);
			try {
				FileWriter tmpFile = new FileWriter(theFileW,
				    true);
				for (int i = 1; i <= value; i++) {
					tmpFile.write("@");
				}
				tmpFile.close();

			} catch (IOException e) {
				// TODO Bloc catch auto-généré
				e.printStackTrace();
			}

			break;
		default:
			System.out.println("Invalid Mode : " + mode);
			break;
		}
	}

	/**
	 * Cette méthode permet de faire un accès disque en lecture et
	    écriture.
```

```
         * Elle permet également de simuler l'accès à une Base de Données
         *
         * @param reading
         *            Nombre de lecture
         * @param writing
         *            Nombre d'écriture
         * @return
         */
        public void disk(int reading, int writing) {

                this.disk(reading,'r');
                this.disk(writing,'w');

        }


        /**
         * Cette méthode crée un nom de fichier à utiliser pour l'écriture
         *
         * @param
         *
         * @return
         */
        private void createTempoFile() {

                Date date1 = new Date();
                SimpleDateFormat simpleFormat = new SimpleDateFormat(
                                "dd_MM_yyyy_HH_mm_ss_ms");
                simpleFormat.format(date1);
                String sNomFichier = "tempoFile" + "_" + simpleFormat.format(
                    date1)
                                + ".txt";
                theFileW  = System.getProperty("user.dir") + "\\tmpWriteFile\\
                    "
                                + sNomFichier;

        }
}
```

9.4 Les collections

9.4.1 La classe Machines

```
package collections;

/**
 * @author Jean-Claude Schmidt
 * @version
 * @fundp lihd-m
 */

import factory.Machine;
import java.util.HashMap;
import java.util.Set;

/**
```

```
 * @author  JC
 */
public class Machines {

        /**
         * Machine collection
         */
        static HashMap machineList = new HashMap();

        /**
         * The constructor
         */
        private Machines() {
        }

        public static void addMachine(String id, Machine remoteMachine) {
                machineList.put(id, remoteMachine);
        }

        /**
         * Returns the machine equals to the key
         *
         * @return Returns the machine.
         * @uml.property name="machine"
         */
        public static Machine getMachine(String key) {
                return (Machine) machineList.get(key);
        }

        /**
         * Returns the list of machine
         * @uml.property  name="machineList"
         */
        public static String[] getMachineList() {
                Set s  = machineList.keySet();
                return (String[]) s.toArray();
        }

        /**
         * Clear the collection
         *
         */
        public static void clear() {
                machineList.clear();
        }

}
```

9.4.2 La classe MachineComponents

```
package collections;

import component.Component;
import java.util.ArrayList;
```

```
public class MachineComponents {

        /**
         * @uml.property name="componentList"
         */
        private ArrayList<Component> componentList;

        /**
         * The constructor
         */
        public MachineComponents() {
                componentList = new ArrayList<Component>();

        }

        public void addComponent(Component remoteComponent) {
                componentList.add(remoteComponent);
        }

        /**
         * Returns the component equals to the index
         *
         * @return Returns the component.
         * @uml.property name="component"
         */
        public Component getComponent(int index) {
                return (Component) componentList.get(index);
        }


        /**
         * Returns the size of collection
         *
         */
        public int size() {
                return componentList.size();
        }
        /**
         * Clear the collection
         *
         */
        public void clear() {
                componentList.clear();
        }


}
```

9.4.3 La classe Components

```
package collections;

import component.Component;
import component.Parameters;
import co.StartOneway;
import factory.Machine;
import java.rmi.RemoteException;
import java.util.ArrayList;
```

```
public class Components {

        public ArrayList<Component> collection = new ArrayList<Component>();

        private Machine machine;

        private Parameters parameters;

        private Component component = null;

        private Component nextComponent = null;

        // the constructor
        public Components() {
        }



        /**
         * @param machine
         *            The machine to set
         *
         * @uml.property name="machine"
         */
        public void setMachine(Machine m) {
                this.machine = m;
        }

        /**
         * @param parameters
         *            The parameters to set
         *
         * @uml.property name="parameters"
         */
        public void setParameters(Parameters p) {
                this.parameters = p;
        }

        public void connect(Component nextC) {
                for (int i = 0; i < collection.size(); i++) {
                        component = (Component) collection.get(i);
                        try {
                                // invoke the method connect() on the remote
                                    object "get(i)"
                                component.connect(nextC);
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }
        }

        public void connect(int nextC) {
                nextComponent = (Component) collection.get(nextC - 1);
                for (int i = 0; i < collection.size(); i++) {
                        component = (Component) collection.get(i);
                        try {
```

```
                        // invoke the method connect() on the remote
                            object "get(i)"
                        component.connect(nextComponent);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }

            }
    }

    public void connect(int firstC, int secondC) {

            component = (Component) collection.get(firstC - 1);
            nextComponent = (Component) collection.get(secondC - 1);
            try {
                    // invoke the method connect() on the remote object "
                        component"
                    component.connect(nextComponent);
            } catch (RemoteException e) {
                    e.printStackTrace();
            }
    }



    public void start() {

            for (int i = 0; i < collection.size(); i++) {
                    component = (Component) collection.get(i);
                    try {
                        // invoke the method start() on the remote
                            object "get(i)"
                        component.start(parameters);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }
            }
    }

    public void startOneway() {

            for (int i = 0; i < collection.size(); i++) {
                    component = (Component) collection.get(i);
                    new StartOneway(component, parameters);
            }
    }

    public void startOneway(int i) {

            component = (Component) collection.get(i - 1);
            new StartOneway(component, parameters);
    }

    public void startOneway(int i, int j) {

            for (int k = i; k <= j; k++) {
                    component = (Component) collection.get(k - 1);
                    new StartOneway(component, parameters);
```

```
                }
        }


        public void start(int i) {

                component = (Component) collection.get(i - 1);
                try {
                        // invoke the method start() on the remote object "get
                            (i-1)"
                        component.start(parameters);
                } catch (RemoteException e) {
                        e.printStackTrace();
                }
        }

        public void start(int i, int j) {

                for (int k = i; k <= j; k++) {
                        component = (Component) collection.get(k - 1);
                        try {
                                // invoke the method start() on the remote
                                    object "get(k-1)"
                                component.start(parameters);
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }
        }

        public Component get(int i) {
                // return the reference of the remote object "get(i-1)"
                return (Component) collection.get(i - 1);
        }

        public void stop() {

                for (int i = 0; i < collection.size(); i++) {
                        component = (Component) collection.get(i);
                        try {
                                // invoke the method stop() on the remote
                                    object "get(i)"
                                component.stop();
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }
        }

        public void stop(int i) {

                component = (Component) collection.get(i - 1);
                try {
                        // invoke the method stop() on the remote object "get(
                            i-1)"
                        component.stop();
                } catch (RemoteException e) {
                        e.printStackTrace();
```

```
                }
        }

        public void stop(int i, int j) {

                for (int k = i; k <= j; k++) {
                        component = (Component) collection.get(k - 1);
                        try {
                                // invoke the method stop() on the remote
                                    object "get(k-1)"
                                component.stop();
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                }
        }

}
```

9.5 Le système de mesure

9.5.1 La classe Measure et ses classes dérivées

```
/**
 *
 */
package measure;

import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

import component.GenericComponent;

/**
 * @author Schmidt JC
 */
public abstract class Measure {

        /**
         * The value of the mesure
         *
         * @uml.property name="value"
         */
        public long value = 0;

        /**
         * The unit of the mesure (ms, sec, %, Mb, ...)
         */
        public String unit = "";

        /**
         * Flag indicating if the mesure is terminated. This flag is required
             for
         * the save() method.
         */
```

```
        boolean terminated = false;

        public String measureName = "";

        public String componentName = "";

        public String measureType = "";

        public String resultFileW = "";

        public Measure() {

        }

        /**
         * Start the measure.
         */
        public abstract void start();

        /**
         * Stop the measure
         */
        public void stop() {
                terminated = true;
        }

        public void save() {
                // Open and writing in the file 'resultFileW'
                try {
                        FileWriter resFile = new FileWriter(resultFileW, true)
                            ;

                        resFile.write(componentName + ";" + measureName + ";"
                                        + measureType + ";" + unit + ";" +
                                            value + "\n");

                        resFile.close();

                } catch (IOException e) {
                        // TODO Bloc catch auto-généré
                        e.printStackTrace();
                }
        }

        /**
         * Affiche la mesure sur la console système
         */
        public void display() {
                System.out.println(componentName + ";" + measureName + ";"
                                + measureType + ";" + unit + ";" + value);
        }

        /**
         * @return Boolean
         * @uml.property name="terminated"
         */
        public boolean isTerminated() {
                return terminated;
```

```
        }

        /**
         * @return Float
         * @uml.property name="value"
         */
        public long getValue() {
                return value;

        }

        /**
         * Setter of the property <tt>value</tt>
         *
         * @param value
         *            The value to set.
         * @uml.property name="value"
         */
        public void setValue(long value) {
                this.value = value;
        }



}
```

```
package measure;

import component.GenericComponent;


public class Chrono extends measure.Measure {



        /**
         * The constructor
         *
         */
        public Chrono(GenericComponent c, String measureName) {
                unit = "ms";
                measureType = "Chrono";
                this.componentName = c.machineID+";"+c.componentName;
                this.measureName = measureName;
                this.resultFileW = c.resultFileW;
        }
        /**
         * Get the current time in millisecond
         *
         */
        public void start() {
                 value = java.lang.System.currentTimeMillis() ;
        }

        /**
         * Get the current time and substract the old value for compute the
             total time
         *
         */
```

```
        public void stop() {
                value = java.lang.System.currentTimeMillis() - value;
        }


}
```

```
package measure;

import component.GenericComponent;

public class Counter extends measure.Measure {



        /**
         * The constructor
         *
         */
        public Counter(GenericComponent c, String measureName) {
                value = 0;
                unit = "";
                measureType = "Counter";

                this.componentName = c.machineID+";"+c.componentName;
                this.measureName = measureName;
                this.resultFileW = c.resultFileW;
        }

        /**
         * Initialise la mesure
         */
        public void start() {
                value = 0;

        }

        /**
         *
         * Arrête la mesure
         */
        public void stop() {

        }

        /**
         * Méthode qui incrémente le compteur
         */
        public void increment() {

                value = value + 1;
        }



}
```

```
package measure;
```

```
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;

import component.GenericComponent;


public class Memory extends measure.Quantity {

        public String memUsed = "";


        public Memory(GenericComponent c, String measureName) {

                unit = "Bytes";
                measureType = "Memory";

                this.componentName = c.machineID+";"+c.componentName;
                this.measureName = measureName;
                this.resultFileW = c.resultFileW;
        }

        public void start() {

        }

        /**
         * Get the current memory usage and set the used memory to the field '
             value'.
         */
        public void stop() {
                MemoryMXBean memoryBean = ManagementFactory.getMemoryMXBean();
                memUsed = memoryBean.getHeapMemoryUsage().toString();
                String mem1 = memUsed.substring(memUsed.indexOf("used = ")+7,
                    memUsed.indexOf(" committed"));
                String mem2 = mem1.substring(0,mem1.indexOf("("));
                // Conversion de String en Long
                value = Long.parseLong(mem2);
        }


}
```

9.6 Les classes utiles aux composants

9.6.1 L'interface Component

```
package component;

import java.io.Serializable;
import java.rmi.Remote;
import java.rmi.RemoteException;


/**
 * @author JC
 *
```

```
 */
public interface Component extends Remote {

    public void          start(Params p) throws RemoteException;
    public void          stop() throws RemoteException;
    public void          connect(Component comp) throws RemoteException;
    public void          request(String s) throws RemoteException;
    public String        requestS(String s) throws RemoteException;

}
```

9.6.2 La classe Parameters et son interface

```
package co;

import java.util.Hashtable;


public class Parameters implements Params {

    private Hashtable<String, Component> hash = new Hashtable<String,
        Component>();

    public Parameters() {
    }

    public Parameters(String key, Component component) {
        hash.put(key, component);
    }

    public void add(String key, Component component) {
        hash.put(key, component);
    }

    public void clear() {
        hash.clear();
    }

    public Component resolve(String key) throws java.rmi.RemoteException {
        return (Component) hash.get(key);
    }




}
```

```
package co;

import java.io.Serializable;



public interface Params extends Serializable {

    public Component resolve(String key) throws java.rmi.RemoteException;
```

```
}
```

9.6.3 La super-classe d'un composant

```
package component;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.text.SimpleDateFormat;
import java.util.Date;

import tools.Simul;

import measure.Counter;

public class GenericComponent extends UnicastRemoteObject implements Component
    {

        public boolean          RUNNING = false;
        public Component        nextComponent = null;
        public String           componentName = "";
        public int                      componentID = 0;
        public String           machineID = "";
        public Params           parameters = null;
        public Simul            simul;

        public Counter          cptRequest;
        public String           resultFileW = "";



    public GenericComponent() throws RemoteException {
        super();

        simul = new Simul();
        createResultFile();

    }

    public void connect(Component comp) throws RemoteException {
        this.nextComponent = comp;
    }

    public void start(Params p) throws RemoteException {
        RUNNING = true;
        parameters = p;
        System.out.println(machineID+"."+componentName+".start()___RUNNING_=_
            " + RUNNING);
        cptRequest = new Counter(this,"CptRequest");
    }

    public void stop() throws RemoteException {
        RUNNING = false;
        System.out.println(machineID+"."+componentName+".stop()___RUNNING_=_"
             + RUNNING);

    }
```

```
        public void request(String s) throws RemoteException {
                cptRequest.increment();
        }

        public String requestS(String s) throws RemoteException {
                return s;

        }

        private void createResultFile() {

                Date date1 = new Date();
                SimpleDateFormat simpleFormat = new SimpleDateFormat(
                                "dd_MM_yyyy_HH_mm_ss_ms_SSSZ");
                simpleFormat.format(date1);
                String sNomFichier = "ResultFile_" + simpleFormat.format(date1
                    )
                                + ".csv";
                resultFileW  = System.getProperty("user.dir") + "\\ResultFiles
                    \\"
                                + sNomFichier;

        }
}
```

Index